36th International Symposium on Distributed Computing

DISC 2022, October 25–27, 2022, Augusta, Georgia, USA

Edited by

Christian Scheideler

Editors

Christian Scheideler
Universität Paderborn, Germany
scheideler@upb.de

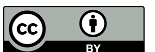
ACM Classification 2012
Software and its engineering → Distributed systems organizing principles; Computing methodologies → Distributed computing methodologies; Computing methodologies → Concurrent computing methodologies; Hardware → Fault tolerance; Information systems → Data structures; Networks; Theory of computation; Theory of computation → Models of computation; Theory of computation → Design and analysis of algorithms

ISBN 978-3-95977-255-6

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-255-6.

Publication date
October, 2022

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

License

This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0): https://creativecommons.org/licenses/by/4.0/legalcode.
In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:
- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.DISC.2022.0

ISBN 978-3-95977-255-6 **ISSN 1868-8969** **https://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board

- Luca Aceto (*Chair*, Reykjavik University, IS and Gran Sasso Science Institute, IT)
- Christel Baier (TU Dresden, DE)
- Mikolaj Bojanczyk (University of Warsaw, PL)
- Roberto Di Cosmo (Inria and Université de Paris, FR)
- Faith Ellen (University of Toronto, CA)
- Javier Esparza (TU München, DE)
- Daniel Král' (Masaryk University - Brno, CZ)
- Meena Mahajan (Institute of Mathematical Sciences, Chennai, IN)
- Anca Muscholl (University of Bordeaux, FR)
- Chih-Hao Luke Ong (University of Oxford, GB)
- Phillip Rogaway (University of California, Davis, US)
- Eva Rotenberg (Technical University of Denmark, Lyngby, DK)
- Raimund Seidel (Universität des Saarlandes, Saarbrücken, DE and Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, DE)

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Preface
Christian Scheideler 0:ix

Organization
.... 0:xi

Awards
.... 0:xv

2022 Edsger W. Dijkstra Prize in Distributed Computing
.... 0:xvii

Principles of Distributed Computing Doctoral Dissertation Award
.... 0:xix

Invited Talks

Graph Coloring, Palette Sparsification, and Beyond
Sepehr Assadi 1:1–1:1

Managing the Cyber Risk in a Decoupled World: Does This Bring Potential Opportunities in Computer Science?
Roberto Baldoni 2:1–2:1

Using Linearizable Objects in Randomized Concurrent Programs
Jennifer L. Welch 3:1–3:1

Regular Papers

Good-Case Early-Stopping Latency of Synchronous Byzantine Reliable Broadcast: The Deterministic Case
Timothé Albouy, Davide Frey, Michel Raynal, and François Taïani 4:1–4:22

Polynomial-Time Verification and Testing of Implementations of the Snapshot Data Structure
Gal Amram, Avi Hayoun, Lior Mizrahi, and Gera Weiss 5:1–5:20

Almost Universally Optimal Distributed Laplacian Solvers via Low-Congestion Shortcuts
Ioannis Anagnostides, Christoph Lenzen, Bernhard Haeupler, Goran Zuzic, and Themis Gouleakis 6:1–6:20

Byzantine Connectivity Testing in the Congested Clique
John Augustine, Anisur Rahaman Molla, Gopal Pandurangan, and Yadu Vasudev 7:1–7:21

Efficient Classification of Locally Checkable Problems in Regular Trees
Alkida Balliu, Sebastian Brandt, Yi-Jun Chang, Dennis Olivetti, Jan Studený, and Jukka Suomela 8:1–8:19

36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Exponential Speedup over Locality in MPC with Optimal Memory
Alkida Balliu, Sebastian Brandt, Manuela Fischer, Rustam Latypov, Yannic Maus, Dennis Olivetti, and Jara Uitto 9:1–9:21

Holistic Verification of Blockchain Consensus
Nathalie Bertrand, Vincent Gramoli, Igor Konnov, Marijana Lazić, Pierre Tholoniat, and Josef Widder 10:1–10:24

How to Meet at a Node of Any Connected Graph
Subhash Bhagat and Andrzej Pelc 11:1–11:16

Liveness and Latency of Byzantine State-Machine Replication
Manuel Bravo, Gregory Chockler, and Alexey Gotsman 12:1–12:19

Oracular Byzantine Reliable Broadcast
Martina Camaioni, Rachid Guerraoui, Matteo Monti, and Manuel Vidigueira 13:1–13:19

Byzantine Consensus Is $\Theta(n^2)$: The Dolev-Reischuk Bound Is Tight Even in Partial Synchrony!
Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira 14:1–14:21

Dynamic Probabilistic Input Output Automata
Pierre Civit and Maria Potop-Butucaru 15:1–15:18

How to Wake up Your Neighbors: Safe and Nearly Optimal Generic Energy Conservation in Radio Networks
Varsha Dani and Thomas P. Hayes 16:1–16:22

Contention Resolution Without Collision Detection: Constant Throughput *And* Logarithmic Energy
Gianluca De Marco, Dariusz R. Kowalski, and Grzegorz Stachowiak 17:1–17:21

Smoothed Analysis of Information Spreading in Dynamic Networks
Michael Dinitz, Jeremy Fineman, Seth Gilbert, and Calvin Newport 18:1–18:22

An Almost Singularly Optimal Asynchronous Distributed MST Algorithm
Fabien Dufoulon, Shay Kutten, William K. Moses Jr., Gopal Pandurangan, and David Peleg 19:1–19:24

Locally Restricted Proof Labeling Schemes
Yuval Emek, Yuval Gil, and Shay Kutten 20:1–20:22

Distributed Construction of Lightweight Spanners for Unit Ball Graphs
David Eppstein and Hadi Khodabandeh 21:1–21:23

Improved Deterministic Connectivity in Massively Parallel Computation
Manuela Fischer, Jeff Giliberti, and Christoph Grunau 22:1–22:17

Fault Tolerant Coloring of the Asynchronous Cycle
Pierre Fraigniaud, Patrick Lambein-Monette, and Mikaël Rabie 23:1–23:22

Distributed Randomness from Approximate Agreement
Luciano Freitas, Petr Kuznetsov, and Andrei Tonkikh 24:1–24:21

Fragmented ARES: Dynamic Storage for Large Objects
Chryssis Georgiou, Nicolas Nicolaou, and Andria Trigeorgi 25:1–25:24

Fast Distributed Vertex Splitting with Applications
Magnús M. Halldórsson, Yannic Maus, and Alexandre Nolin 26:1–26:24

Broadcast CONGEST Algorithms Against Eavesdroppers
Yael Hitron, Merav Parter, and Eylon Yogev 27:1–27:19

Routing Schemes and Distance Oracles in the Hybrid Model
Fabian Kuhn and Philipp Schneider 28:1–28:22

On Payment Channels in Asynchronous Money Transfer Systems
Oded Naor and Idit Keidar 29:1–29:20

The Space Complexity of Scannable Objects with Bounded Components
Sean Ovens 30:1–30:18

Near-Optimal Distributed Computation of Small Vertex Cuts
Merav Parter and Asaf Petruschka 31:1–31:21

Õptimal Dual Vertex Failure Connectivity Labels
Merav Parter and Asaf Petruschka 32:1–32:19

Safe Permissionless Consensus
Youer Pu, Lorenzo Alvisi, and Ittay Eyal 33:1–33:15

Packet Forwarding with a Locally Bursty Adversary
Will Rosenbaum 34:1–34:18

The Weakest Failure Detector for Genuine Atomic Multicast
Pierre Sutra 35:1–35:19

On Implementing SWMR Registers from SWSR Registers in Systems with Byzantine Failures
Xing Hu and Sam Toueg 36:1–36:19

Space-Stretch Tradeoff in Routing Revisited
Anatoliy Zinovyev 37:1–37:16

Brief Announcements

Brief Announcement: Authenticated Consensus in Synchronous Systems with Mixed Faults
Ittai Abraham, Danny Dolev, Alon Kagan, and Gilad Stern 38:1–38:3

Brief Announcement: It's not easy to relax: liveness in chained BFT protocols
Ittai Abraham, Natacha Crooks, Neil Giridharan, Heidi Howard, and Florian Suri-Payer 39:1–39:3

Brief Announcement: Distributed Algorithms for Minimum Dominating Set Problem and Beyond, a New Approach
Sharareh Alipour and Mohammadhadi Salari 40:1–40:3

Brief Announcement: Survey of Persistent Memory Correctness Conditions
Naama Ben-David, Michal Friedman, and Yuanhao Wei 41:1–41:4

Brief Announcement: Minimizing Congestion in Hybrid Demand-Aware Network Topologies
Wenkai Dai, Michael Dinitz, Klaus-Tycho Foerster, and Stefan Schmid 42:1–42:3

Brief Announcement: Computing Power of Hybrid Models in Synchronous Networks
Pierre Fraigniaud, Pedro Montealegre, Pablo Paredes, Ivan Rapaport, Martín Ríos-Wilson, and Ioan Todinca 43:1–43:3

Brief Announcement: New Clocks, Fast Line Formation and Self-Replication Population Protocols
Leszek Gąsieniec, Paul Spirakis, and Grzegorz Stachowiak 44:1–44:3

Brief Announcement: Performance Anomalies in Concurrent Data Structure Microbenchmarks
Rosina F. Kharal and Trevor Brown 45:1–45:3

Brief Announcement: Gathering Despite Defected View
Yonghwan Kim, Masahiro Shibata, Yuichi Sudo, Junya Nakamura, Yoshiaki Katayama, and Toshimitsu Masuzawa 46:1–46:3

Brief Announcement: An Effective Geometric Communication Structure for Programmable Matter
Irina Kostitsyna, Tom Peters, and Bettina Speckmann 47:1–47:3

Brief Announcement: Distributed Quantum Interactive Proofs
François Le Gall, Masayuki Miyamoto, and Harumichi Nishimura 48:1–48:3

Brief Announcement: Null Messages, Information and Coordination
Raïssa Nataf, Guy Goren, and Yoram Moses 49:1–49:3

Brief Announcement: Asymmetric Mutual Exclusion for RDMA
Jacob Nelson-Slivon, Lewis Tseng, and Roberto Palmieri 50:1–50:3

Brief Announcement: Foraging in Particle Systems via Self-Induced Phase Changes
Shunhao Oh, Dana Randall, and Andréa W. Richa 51:1–51:3

Brief Announcement: Temporal Locality in Online Algorithms
Maciej Pacut, Mahmoud Parham, Joel Rybicki, Stefan Schmid, Jukka Suomela, and Aleksandr Tereshchenko 52:1–52:3

Preface

Welcome to DISC 2022, the 36th International Symposium on Distributed Computing, held on October 25–27, 2022, in Augusta, Georgia, USA. DISC is an international forum on the theory, design, analysis, and implementation of distributed systems and networks, focusing on distributed computing in all its forms. DISC is organized in cooperation with the European Association for Theoretical Computer Science (EATCS).

This volume contains the papers presented at DISC 2022. We received 117 regular paper submissions and 2 brief announcement submissions. The program committee consisted of 34 members, and the committee was assisted by 110 external reviewers. All submissions were evaluated by at least three reviewers. The program committee used a relaxed form of double-blind review: the submissions were anonymous but the authors were allowed to disseminate their work through arXiv or other online repositories and to give presentations of their works. Final decisions were made during a virtual PC meeting. We accepted 34 regular papers (an acceptance rate of 29%) and 15 brief announcements for presentation at DISC 2022. The keynotes were given by Roberto Baldoni, Jennifer Welch, and Sepehr Assadi.

This volume also includes the citations for the best paper and best student paper awards at DISC 2022, as well as citations for two awards jointly sponsored by DISC and the ACM Symposium on Principles of Distributed Computing (PODC):

- The 2022 Edsger W. Dijkstra Prize in Distributed Computing was presented at PODC 2022 to Maged M. Michael for his paper "Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes" and to Maurice Herlihy, Victor Luchangco, and Mark Moir for their paper "The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures."
- The 2022 Principles of Distributed Computing Doctoral Dissertation Award was presented at DISC 2022 to Dr. Naama Ben-David for her dissertation "Theoretical Foundations for Practical Concurrent and Distributed Computation" and to Dr. Manuela Fischer for her dissertation "Local Algorithms for Classic Graph Problems."

I would like to thank everyone who contributed to DISC 2022: the authors of the submitted papers, PC members and external reviewers, keynote speakers, members of the organizing committee, workshop organizers, members of the award committees, and participants at the conference. I would also like to thank the members of the steering committee, former chairs and many other members of the community for their valuable assistance and suggestions, EATCS for their support, and the staff at Schloss Dagstuhl – Leibniz-Zentrum für Informatik for their help in preparing these proceedings.

October 2022

Christian Scheideler
DISC 2022 Program Chair

36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Organization

DISC, the International Symposium on Distributed Computing, is an annual forum for presentation of research on all aspects of distributed computing. It is organized in cooperation with the European Association for Theoretical Computer Science (EATCS). The symposium was established in 1985 as a biannual International Workshop on Distributed Algorithms on Graphs (WDAG). The scope was soon extended to cover all aspects of distributed algorithms and WDAG came to stand for International Workshop on Distributed AlGorithms, becoming an annual symposium in 1989. To reflect the expansion of its area of interest, the name was changed to DISC (International Symposium on DIStributed Computing) in 1998, opening the symposium to all aspects of distributed computing. The aim of DISC is to reflect the exciting and rapid developments in this field.

Program Chair

Christian Scheideler	Paderborn University (Germany)

Program Committee

John Augustine	IIT Madras (India)
Naama Ben David	VMware Research (USA)
Borzoo Bonakdarpour	Michigan State University (USA)
Marthe Bonamy	University of Bordeaux (France)
Christian Cachin	University of Bern (Switzerland)
Artur Czumaj	University of Warwick (UK)
Joshua Daymude	Arizona State University (USA)
Michal Dory	ETH Zürich (Switzerland)
Robert Elsässer	University of Salzburg (Austria)
Laurent Feuilloley	University of Lyon (France)
Michele Flammini	Gran Sasso Science Institute (Italy)
Paola Flocchini	University of Ottawa (Canada)
Davide Frey	Inria centre at Rennes University (France)
Luisa Gargano	University of Salerno (Italy)
Magnús M. Halldórsson	Reykjavik University (Iceland)
Mohammad Taghi Hajiaghayi	University of Maryland (USA)
Alex Kogan	Oracle Labs (USA)
Shay Kutten	Technion (Israel)
Thomas Locher	DFINITY Foundation (Switzerland)
Victor Luchangco	Algorand (USA)
Yannic Maus	TU Graz (Austria)
Othon Michail	University of Liverpool (UK)
Boaz Patt-Shamir	Tel Aviv University (Israel)
David Peleg	Weizmann Institute (Israel)
Maria Potop-Butucaru	Sorbonne University (France)
Sergio Rajsbaum	UNAM (Mexico)
Peter Robinson	Augusta University (USA)

Jared Saia	University of New Mexico (USA)
Christian Scheideler (Chair)	Paderborn University (Germany)
Stefan Schmid	TU Berlin (Germany)
Gokarna Sharma	Kent State University (USA)
Jukka Suomela	Aalto University (Finland)
Yukiko Yamauchi	Kyushu University (Japan)
Haifeng Yu	National University of Singapore (Singapore)

Organizing Committee

Bogdan Chlebus	Augusta University (USA)
Darek Kowalski	Augusta University (USA)
Yannic Maus (Workshops Chair)	TU Graz (Austria)
Regina White (Business Manager)	Augusta University (USA)
Caroline Eaker	Augusta University (USA)
Dennis Olivetti (DISC Webmaster)	Gran Sasso Science Institute (Italy)
Reza Rahaeimehr (Local Web Co-Chair)	Augusta University (USA)
Peter Robinson (Workshop Coordinator)	Augusta University (USA)
Alex Schwarzmann (Chair)	Augusta University (USA)
Edward Tremel (Local Web Co-Chair)	Augusta University (USA)
Steve Weldon (Local Arr.)	Augusta University (USA)
Costas Busch (Student Grants Chair)	Augusta University (USA)

Steering Committee

Hagit Attiya	Technion (Israel)
Seth Gilbert	National University of Singapore (Singapore)
Moti Medina	Bar-Ilan University (Israel)
Calvin Newport	Georgetown University (USA)
Andréa Richa (Chair)	Arizona State University (USA)
Christian Scheideler	Paderborn University (Germany)
Jukka Suomela (Vice Chair)	Aalto University (Finland)

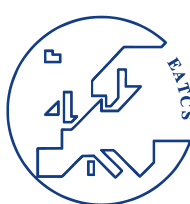
External Reviewers

Ittai Abraham	Duncan Adamson	Vitaly Aksenov
Nada Almalki	Abdullah Almethen	Orestis Alpos
Ignacio Amores-Sesar	Paul Attie	Philipp Bamberger
Kiarash Banihashem	Gregor Bankhamer	Nicolas Bousquet
Manuel Bravo	Trevor Brown	Soumyottam Chatterjee
Gregory Chockler	Matthew Connor	Gennaro Cordasco
Sam Coy	Emilio Cruciani	Varsha Dani
Paolo D'Arco	Peter Davies	Yoann Dieudonne
Oyendrila Dobe	Fabien Dufoulon	Mahsa Eftekhari
Ryota Eguchi	Giovanni Farina	Roberto Ferrara
Orr Fischer	Matthias Függer	Ritam Ganguly

Chryssis Georgiou	Jacob Gilbert	Robert Gmyr
Emmanuel Godard	Guy Goren	Thorsten Götte
Samira Goudarzi	Christoph Grunau	Chetan Gupta
Yael Hitron	Tzu-Han Hsu	Mohammad Abirul Islam
Taisuke Izumi	Peyman Jabbarzade	William K. Moses Jr.
Eleni Kanellou	Mahimna Kelkar	Yonghwan Kim
Valerie King	Marina Knittel	Dariusz Kowalski
Rustam Latypov	David Lehnherr	Dean Leitersdorf
Stefano Leucci	David Liedtke	Giuseppe Antonio Di Luna
Nancy Lynch	Gianluca De Marco	Giovanna Melideo
Darya Melnyk	Jovana Micic	Gopinath Mishra
Slobodan Mitrović	Mark Moir	Anisur Rahaman Molla
Pedro Montealegre	Anish Mukherjee	Junya Nakamura
Alfredo Navarra	Alexandre Nolin	Krzysztof Nowicki
Dennis Olivetti	Jan Olkowski	Krzysztof Onak
Fukuhito Ooshita	Andreas Padalkin	Shreyas Pai
Gopal Pandurangan	Ashish Parihar	Merav Parter
Sathya Peri	Yvonne-Anne Pignolet	Julian Portmann
Srikkanth Ramachandran	Srivatsan Ravi	Adele Rescigno
Etienne Rivière	Nicola Santoro	Elad Schiller
Philipp Schneider	Michele Scquizzato	Yangguang Shi
George Skretas	Alberto Sonnino	Yuichi Sudo
Aishwarya Thiruvengadam	Giovanni Viglietta	Tijn de Vos
Daniel Warner	Yuanhao Wei	Julian Werthmann
Sravya Yandamuri	Sheng Yang	Maxwell Young
Luca Zanolini	Goran Zuzic	

Acknowledgements

DISC 2022 acknowledges the use of Easychair for handling submissions and managing the review process and LIPIcs for producing and publishing the proceedings. DISC 2022 sponsors are:



DISC is organized in cooperation with the European Association for Theoretical Computer Science (EATCS).

Awards

Best Paper

The DISC Program Committee has selected the following paper to receive the DISC 2022 best paper award:

Smoothed Analysis of Information Spreading in Dynamic Networks
by Michael Dinitz, Jeremy Fineman, Seth Gilbert, and Calvin Newport.

This paper applies smoothed analysis to the study of k-message broadcast in dynamic networks. It shows that even with a small amount of smoothing, a simple distributed random broadcast strategy can significantly outperform the existing worst-case lower bounds. In addition to that, it proves that in static networks the runtime of this strategy further improves, establishing that even in the context of smoothing, changing topologies remain more difficult to move information through than their static counterparts. Interestingly, the tools developed for these analyses can be applied to improve the best-known bounds for k-message broadcast, without smoothing, in the well-mixed dynamic network setting.

Best Student Paper

The DISC Program Committee has selected the following two papers to receive the DISC 2022 best student paper award:

Polynomial-Time Verification and Testing of Implementations of the Snapshot Data Structure
by Gal Amram, Avi Hayoun, Lior Mizrahi and Gera Weiss

and

Byzantine Consensus Is $\Theta(n^2)$: The Dolev-Reischuk Bound Is Tight Even in Partial Synchrony!
by Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira.

The first paper focuses on the correctness of implementations of the snapshot data structure in terms of linearizability and shows that such implementations can be verified in polynomial time. This presents a significant improvement considering that verifying linearizability of implementations of concurrent data structures, in general, is EXPSPACE-complete in the number of program states, and testing linearizability is NP-complete in the length of the tested execution.

The second paper presents SQUAD, a partially synchronous Byzantine consensus protocol with quadratic worst-case communication complexity. The Dolev-Reischuk bound says that any deterministic Byzantine consensus protocol has at least quadratic communication complexity in the worst-case, but a protocol with such a complexity was only known for synchronous environments and the previously best protocols for partial synchrony had a cubic communication complexity.

36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2022 Edsger W. Dijkstra Prize in Distributed Computing

The Edsger W. Dijkstra Prize in Distributed Computing is awarded for outstanding papers on the principles of distributed computing, whose significance and impact on the theory or practice of distributed computing have been evident for at least a decade. It is sponsored jointly by the ACM Symposium on Principles of Distributed Computing (PODC) and the EATCS Symposium on Distributed Computing (DISC). The prize is presented annually, with the presentation taking place alternately at PODC and DISC.

The 2022 Edsger W. Dijkstra Prize in Distributed Computing has been awarded to the papers

- **Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes**, by Maged M. Michael (Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing, PODC 2002, pages 21–30), and
- **The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures**, by Maurice Herlihy, Victor Luchangco, and Mark Moir (Proceedings of the 16th International Symposium on Distributed Computing, DISC 2002, pages 339–353).

for providing the first general approach to memory reclamation in nonblocking data structures, with significant impact both in research and practice.

Nonblocking concurrent data structures are central to the theory and practice of parallel programming. Unfortunately, correct nonblocking structures are difficult to write. A key challenge is to ensure, before reclaiming an unlinked node, that no still-active operation retains a reference to that node.

Hazard Pointers, also known as the “Pass the Buck” solution to the Repeat Offenders Problem, were the first general-purpose solution to the memory management problem in nonblocking concurrent data structures. They remain the dominant solution today. By maintaining a global set of references to objects (nodes) to which active threads hold private references, they enable nonblocking code to determine when a node can safely be reclaimed. By organizing the set as a collection of per-thread structures, each of which is accessed primarily by its owner, they avoid most nonlocal references, keeping overhead low enough for all but the most demanding applications.

Together, the winning papers revolutionized both the theory and practice of nonblocking algorithms. They have inspired dozens of follow-on projects and well over a thousand citations. Today, hazard pointers are used in an enormous variety of commercial libraries and systems, making them essential to most of the world’s data centers and multicore devices.

2022 Award Committee:

Marcos Aguilera, VMware Research
Andrea Richa, Arizona State University
Alexander Schwarzmann, Augusta University
Alessandro Panconesi, Sapienza Università Di Roma
Christian Scheideler (chair), Universität Paderborn
Philipp Woelfel, University of Calgary

36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2022 Principles of Distributed Computing Doctoral Dissertation Award

Many exceptionally high-quality doctoral dissertations were submitted for the 2022 Principles of Distributed Computing Doctoral Dissertation Award. After careful long deliberation, the award committee decided to share the award among two:

- **Dr. Naama Ben-David** for her dissertation "Theoretical Foundations for Practical Concurrent and Distributed Computation."
- **Dr. Manuela Fischer** for her dissertation "Local Algorithms for Classic Graph Problems."

Dr. Naama Ben-David completed her thesis on July 22nd, 2020, under the supervision of Prof. Guy E. Blelloch, at Carnegie Mellon University. In her thesis, Dr. Ben-David addressed three modern technologies that play a significant role in concurrent/distributed computing and carefully developed faithful, clean, and theoretically elegant models for each. Based on these models and theories she then developed for each a new distributed algorithm, showed the algorithms applicability in practice, and finally developed practical tools to enable practitioners and theoretical researchers to analyze these systems, and the benefits of the new models and algorithms. The three technologies which Dr. Ben-David addressed in her thesis are: (1) remote direct memory access (RDMA) as a means to share memory among message- passing communicating processors whether in a large network or in a data center, (2) non-volatile random access memories (NVRAM) for which she developed a general simulation that can adapt many classic concurrent algorithms to a setting in which processes using NVRAM can recover after a system fault, and (3) shared-memory concurrent access where Dr. Ben-David developed new careful analysis reflecting their performance in practice.

Dr. Manuela Fischer completed her thesis on 14th June, 2021, under the supervision of Prof. Mohsen Ghaffari, at ETH Zurich. Dr. Fischer's thesis contains a large consistent set of outstanding achievements in the design of distributed algorithms for numerous graph problems in models such as LOCAL and CONGESTED-CLIQUE, with strong connections to the MPC model for parallel computing. Dr. Fischer introduced new techniques for distributed algorithm design, as well as for the analysis of randomized algorithms for graph problems. Two of the results presented in her thesis resolve central problems that were open for over 25 years: the first efficient deterministic distributed algorithm for $(2\Delta - 1)$-edge-coloring, and a tight analysis of a randomized greedy MIS algorithm. The thesis is technically broad and deep, spanning at least five intrinsically different technical challenges: (1) rounding for linear programs, (2) bootstrapping Lovasz Local Lemma algorithms, (3) analyzing an intricate stochastic process, (4) bounding convergence of a Markov chain via path coupling, and (5) a number of randomized ideas in massively parallel computation. Dr. Fischer's techniques have already been adopted by the leading researchers in the field, and are used in several follow-up works.

The award is sponsored jointly by the ACM Symposium on Principles of Distributed Computing (PODC) and the EATCS Symposium on Distributed Computing (DISC). It is presented annually, with the presentation taking place alternately at PODC and DISC. This year it will be presented at DISC, to be held at Augusta, Georgia USA, October 25-27, 2022.

36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The 2022 Principles of Distributed Computing Doctoral Dissertation Award Committee

Yehuda Afek (chair), Tel-Aviv University
Keren Censor-Hillel, Technion
Pierre Fraigniaud, CNRS and Université Paris Cité
Seth Gilbert, National University of Singapore
Gopal Pandurangan, University of Houston
Gadi Taubenfeld, Reichman University, Herzliya

Graph Coloring, Palette Sparsification, and Beyond

Sepehr Assadi
Dept. of Computer Science, Rutgers University, New Jersey, USA

Abstract

Graph coloring is a central problem in graph theory and has numerous applications in diverse areas of computer science. An important and well-studied case of graph coloring problems is the $(\Delta + 1)$ (vertex) coloring problem where Δ is the maximum degree of the graph. Not only does every graph admit a $(\Delta + 1)$ coloring, but in fact we can find one quite easily in linear time and space via a greedy algorithm. But are there more efficient algorithms for $(\Delta + 1)$ coloring that can process massive graphs that even this algorithm cannot handle?

This talk overviews recent results that answer this question in affirmative across a variety of models dedicated to processing massive graphs – streaming, sublinear-time, massively parallel computation, distributed communication, etc. – via a single unified approach: *Palette Sparsification*. We survey the ideas behind these results and techniques, their generalizations to various other coloring problems and even beyond (e.g., to clustering problems), as well as their natural limitations.

The talk is based on a series of joint works with Noga Alon, Andrew Chen, Yu Chen, Sanjeev Khanna, Pankaj Kumar, Parth Mittal, Glenn Sun, and Chen Wang.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Theory of computation → Parallel algorithms; Theory of computation → Distributed algorithms

Keywords and phrases Graph coloring, Palette Sparsification, Sublinear Algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.1

Category Invited Talk

© Sepehr Assadi;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 1; pp. 1:1–1:1
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Managing the Cyber Risk in a Decoupled World: Does This Bring Potential Opportunities in Computer Science?

Roberto Baldoni
National Cybersecurity Agency, Rome, Italy

Abstract

The last thirty years witnessed the growth of both globalization and digital transformation, characterized by information systems becoming interconnected and distributed on a worldwide scale with IT aimed to become a commodity. Cloud computing and blockchain being examples of such robust and distributed technologies which have been the main driver of this globalization process. Global technologies and infrastructures paved the way to organic and highly frequent interactions between millions of companies and organizations in multiple countries almost irrespective of geopolitical implications establishing global and complex interconnected supply chains whose aim was mainly keeping software/devices costs low. This created a virtuous loop that generated an exponential increase of countries' digitalization process and globalized industries.

Like energy, IT progressively became a strategic geopolitical factor as the nation's vital services implementation went digital. As a consequence, governments realized IT cannot be a simple commodity and that they have to manage the cyber risk associated with procured IT in strategic sectors like, for instance, telecommunication, finance and transportation. Governments have to understand and mitigate IT risks coming from these globalized supply chains against operations of potential powerful adversaries. Even a single supply chain dependency can be a risk, also from a national security perspective, when such dependency is established by a provider/vendor under the direct political influence of an untrusted nation or a trusted provider/vendor victim of a state-backed cyber attack. The recent Ukrainian crisis and the large degree of tension between US and China are amplifying risks coming from globalized supply chains in a world that is politically liquid polarizing in at least two blocks.

In addition, the globalization process has shown its natural limits and frailty culminating with the global supply chain crisis created by the effect of the covid-19 pandemic and extreme events due to climate change. Paradoxically, experience shows the main drawback of globalized supply chains is the centralization of certain key manufacturing in restricted geographical areas, this is the case for the infamous chip shortage. This centralization poses risks if a critical portion of these key manufacturing are owned by untrusted actors. A parallel can be seen in the permissionless blockchain technologies based on Proof-of-Work, where the decentralized worldwide spirit has mercilessly converged to a more convenient but weaker almost centralized system which makes it easier for a powerful adversary to take control of the whole blockchain.

The likely trends of the next few years will be a progressive decoupling of supply chains particularly for all software/hardware manufacturing employed into vital services of a nation. This will be a long and non-economically neutral process that will bring in a medium term towards the composition of "friendshoring" or "almost domestic" supply chains where developing robust technologies and algorithms compliant to society values. This is expected to increase the number, the magnitude and complexity of cyber attacks coming from other geopolitical blocks for espionage or terroristic reasons in a continuous hybrid warfare scenario. Computer scientists and engineers will have to cope with the new challenges within this decoupled world. The keynote will be an attempt to shed some light on what this could imply in terms of technology, computing paradigms and nation IT capability.

2012 ACM Subject Classification Security and Privacy; Computing methodologies

Keywords and phrases Supply chain decoupling, technology risk, cyber attacks, computing paradigms and methodologies

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.2

Category Invited Talk

© Roberto Baldoni;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 2; pp. 2:1–2:1
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Using Linearizable Objects in Randomized Concurrent Programs

Jennifer L. Welch
Dept. of Computer Science and Enginering, Texas A&M University, Texas, USA

Abstract

Atomic shared objects, whose operations take place instantaneously, are a powerful technique for designing complex concurrent programs. Since they are not always available, they are typically substituted with software implementations. A prominent condition relating these implementations to their atomic specifications is linearizability, which preserves safety properties of programs using them. However linearizability does not preserve hyper-properties, which include probabilistic guarantees about randomized programs. A more restrictive property, strong linearizability, does preserve hyper-properties but it is impossible to achieve in many situations. In particular, we show that there are no strongly linearizable implementations of multi-writer registers or snapshot objects in message-passing systems. On the other hand, we show that a wide class of linearizable implementations, including well-known ones for registers and snapshots, can be modified to approximate the probabilistic guarantees of randomized programs when using atomic objects.

This is joint work with Hagit Attiya and Constantin Enea.

2012 ACM Subject Classification Theory of computation

Keywords and phrases Concurrent objects, strong linearizability, impossibility proofs, message-passing systems, randomized algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.3

Category Invited Talk

© Jennifer L. Welch;
licensed under Creative Commons License CC-BY 4.0

Good-Case Early-Stopping Latency of Synchronous Byzantine Reliable Broadcast: The Deterministic Case

Timothé Albouy ✉
Univ Rennes, Inria, CNRS, IRISA, France

Davide Frey ✉
Univ Rennes, Inria, CNRS, IRISA, France

Michel Raynal ✉
Univ Rennes, Inria, CNRS, IRISA, France

François Taïani ✉
Univ Rennes, Inria, CNRS, IRISA, France

Abstract

This paper considers the good-case latency of Byzantine Reliable Broadcast (BRB), i.e., the time taken by correct processes to deliver a message when the initial sender is correct, and an essential property for practical distributed systems. Although significant strides have been made in recent years on this question, progress has mainly focused on either asynchronous or randomized algorithms. By contrast, the good-case latency of deterministic synchronous BRB under a majority of Byzantine faults has been little studied. In particular, it was not known whether a good-case latency below the worst-case bound of $t+1$ rounds could be obtained under a Byzantine majority. In this work, we answer this open question positively and propose a deterministic synchronous Byzantine reliable broadcast that achieves a good-case latency of $\max(2, t+3-c)$ rounds, where t is the upper bound on the number of Byzantine processes, and c the number of effectively correct processes.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Reliable Broadcast, Byzantine Faults, Synchronous Systems, Good-case latency, Deterministic Algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.4

Funding This work was partially supported by the French ANR project ByBLoS (ANR-20-CE25-0002-01), and by the PriCLeSS project granted by the Labex CominLabs excellence laboratory of the French ANR (ANR-10-LABX-07-01).

Acknowledgements The authors would like to thank the anonymous reviewers whose careful reading and suggestions helped them improve their paper.

1 Introduction

Introduced in the eighties [14, 20], Byzantine reliable broadcast (BRB) and Byzantine Broadcast (BB) are two fundamental abstractions of distributed computing [5, 7, 9, 10, 19, 22, 23, 26, 25]. BRB assumes that one particular process, the sender, broadcasts a message to the rest of the system and that correct (a.k.a. honest) processes all deliver the value initially broadcast if the sender is correct or that, if it is not, either all agree on some value or none delivers any value. BB further requires that all correct processes always deliver some

© Timothé Albouy, Davide Frey, Michel Raynal, and François Taïani;
licensed under Creative Commons License CC-BY 4.0

value.[1] BRB and BB play a crucial role in many practical distributed applications, from primary-backup state machine replication (SMR) (see, for instance, the discussion in [3]), to broadcast-based money transfer [6, 8, 12, 18].

Good-case latency. In broadcast-based money transfer algorithms, for instance, a cryptocurrency is implemented by merely broadcasting the transfer operations originating from one participant (or in some sharded versions [8] from one *authority*) to the rest of the system. These algorithms do not require consensus, and their performance is directly related to the underlying (Byzantine-tolerant) reliable broadcast algorithm they use. Transfers issued by correct participants are guaranteed to terminate and only involve a single broadcast operation invoked by the issuer. As a result, the latency of these algorithms – as experienced by correct participants – solely depends on the *good-case latency* of the BRB algorithm they use, defined as the time taken for all correct parties to deliver a broadcast message when the initial broadcaster is correct [3]. The *good-case latency* of Byzantine-tolerant broadcast algorithms plays a similarly central role in the performance of SMR algorithms, with vast practical consequences for the performance of BFT replication systems, including consortium [2, 17] and committee-based blockchains [11].

Synchronous networks. In this paper, we focus on the *good-case latency* of BRB algorithms subject to an *arbitrary number of Byzantine failures* (i.e., we assume $n > t$, where n is the number of processes, and t is an upper bound on the number of Byzantine processes). We further assume that processes can use signatures to authenticate messages. We follow in this respect [16] and [27], and in part [3]. Since BRB cannot be solved even in a partially synchronous model when $t \geq n/3$ [15, 20, 24], we also assume a *synchronous* network, in which messages are delivered during the same round in which they are sent. Although synchronous wide-area networks are challenging to realize in practice, they can be approximated with high probability by using sufficiently high timeouts. Synchronous algorithms are further intriguing in their own right and can yield insights into the nature of distributed computing that are useful beyond their specific use.

Randomized synchronous BRB algorithms. The study of randomized synchronous BRB and BB algorithms tolerating arbitrary many Byzantine faults has progressed substantially in recent years [3, 16, 27]. In particular, the solution proposed by Wan, Xiao, Shi, and Devadas [27] and optimized by Abraham, Nayak, Ren, and Xiang [3] presents sublinear worst- and good-case latency bounds in expectation (boiling down to constant numbers of rounds when t, the maximal number of Byzantine processes, is assumed to be a fraction of n). However, these works all rely on *randomization*. Moreover, they generally assume a *weakly adaptive adversary*, an adversary that cannot erase messages sent “just before” a process becomes Byzantine, where “just before” means in the same round. (A notable exception is the solution presented in [26], which tolerates a strongly-adaptive adversary by exploiting time-lock puzzles.) Further, these works do not leverage a lower number of actual faults to provide an *early stopping* property [13]: their latency only involves n, the number of processes, and t, the upper bound on the number of Byzantine processes, but not c, the *effective* number of correct processes. As a result, they cannot exploit a low number of actual failures to provide better latency performance.

[1] In this paper, we will tend to conflate the two problems, as the protocols we discuss solve both BB and BRB.

This paper's contribution. In contrast to randomized solutions, the good-case latency of *deterministic* synchronous BRB and BB algorithms has been little studied. By definition, a deterministic Byzantine-tolerant broadcast algorithm tolerates a strongly adaptive adversary (one that can remove messages "after the fact"). In the worst case, however, its latency is lower-bounded by $t+1$ rounds [14, 15], and optimal algorithms in this respect have been known since the eighties [14, 20].

An unsolved question to this date is thus whether a good-case latency lower than $t+1$ rounds can be achieved using a deterministic algorithm subject to an arbitrary number of Byzantine faults. In this paper, we answer this question positively and propose a deterministic synchronous Byzantine reliable broadcast that achieves a good-case latency of $\mathsf{max}(2, t+3-c)$ rounds, where t is the upper-bound on the number of Byzantine processes, and c the number of effectively correct processes ($c \geq n-t$). The algorithm we propose does not require correct processes to know either n or c. Moreover, and differently from recently proposed solutions to this problem [3, 16, 27], our solution:

- is deterministic (which is why it trivially tolerates a strongly adaptive adversary),
- only relies on signatures, eschewing richer cryptographic primitives (e.g. distributed random coins [16, 27], verifiable random functions [21, 27] or time-lock puzzles [26]),
- ensures delivery in just 2 rounds in good cases as soon as the effective number of correct processes, c, is at least $t+1$, thus improving on all existing solutions.[2]

The early stopping nature of our solution lends it a substantial advantage even when the effective number of correct processes c is less than $t+1$. For instance, assuming $t < 3/4 \times n$, and an intermediate situation where only $\lfloor t/2 \rfloor$ processes have effectively been compromised, the good-case latency of our algorithm outperforms that of the best-known randomized algorithm up to $n \leq 43$, and is at least as good up to $n \leq 51$, making it competitive in a wide range of small- to medium-scale practical distributed systems.

Our algorithm, although not trivial, remains surprisingly simple. It exploits patterns in signature chains, thus extending an idea as old as the problem itself [14, 20].

2 Background and Related Work

The Synchronous Byzantine Reliable Broadcast problem was first introduced in [24] by Lamport, Shostak, and Pease, who proposed in [20] a deterministic solution based on signature chains that requires $t+1$ rounds (both in good and bad cases), where $t < n$ is an upper bound on the number of Byzantine processes present in the system. This worst-case round complexity was shown by Dolev and Strong [14] to be optimal for deterministic algorithms. This result was later refined by Dolev, Reischuk, and Strong who showed that $\mathsf{min}(n-1, n-c+2, t+1)$ rounds are necessary to realize Synchronous Byzantine Broadcast [13], where $c \geq n-t$ is the effective number of correct processes in a given run. They also present in the same paper a deterministic signature-free algorithm that achieves this bound provided that $n > \mathsf{max}(4t, 2t^2 - 2t + 2)$. The salient properties of this algorithm are summarized in the first column of Table 1 and compared to more recent works and to this paper (last column).

In recent years, substantial progress has been achieved to circumvent the hard bound of $t+1$ rounds for deterministic BRB and BB algorithm by exploiting *randomization*, and generally assuming a weakly adaptive adversary, i.e., an adversary that can adaptively corrupt processes, but cannot remove messages sent in the round when a process becomes Byzantine.

[2] More generally, our good-case latency is *early stopping* [13], in that, in good cases, our algorithm will stop earlier when the effective number of correct processes c increases.

Table 1 Assumptions, guarantees, and latencies of synchronous signature-based BRB algorithms (* indicates an expected number of rounds).

	Dolev, Reischuk & Strong [13]	Fitzi & Nielsen [16]	Wan et al. [27] + Abraham et al. [3]	This paper
Deterministic	**yes**	no	no	**yes**
Strong adversary	**yes**	no	no	**yes**
Early stopping	**yes**	no	no	**yes**
Dishonest majority	no	**yes**	**yes**	**yes**
$n >$	$\max(4t, 2t^2-2t+2)$	–	–	–
Worst-case latency	$\min(n-c+2, t+1)$	$\max(7, \lfloor\frac{3t-n}{2}\rfloor+7)+O(1)$ *	$O\big((\frac{n}{n-t})^2\big)$ *	$t+1$
Good-case latency	2	$\max(6, \lfloor\frac{3t-n}{2}\rfloor+6)$	$\lceil\frac{n}{n-t}\rceil+\lfloor\frac{n}{n-t}\rfloor$	$\max(2, t+3-c)$

Assuming a majority of Byzantine processes, Fitzi and Nielsen proposed in [16] a randomized algorithm that achieves Byzantine Agreement in an expected number of $\lfloor(3t-n)/2\rfloor+7+O(1)$ rounds[3], and a good-case latency of $\lfloor(3t-n)/2\rfloor+6$ deterministic rounds.

In 2020, Wan, Xiao, Shi, and Devadas presented a randomized algorithm that achieves BB in $O\big((\frac{n}{n-t})^2\big)$ expected synchronous rounds [27]. Last year, in an in-depth study of the good-case latency of BB and BRB algorithms [3] (extended version in [4]), Abraham, Nayak, Ren, and Xiang proved a lower bound of $\lfloor n/(n-t)\rfloor-1$ rounds for the good-case latency of synchronous BRB. They then explained how the solution presented in [27] can be optimized to deliver a good-case latency of $\lceil n/(n-t)\rceil+\lfloor n/(n-t)\rfloor$ rounds (about $2n/(n-t)\pm 1$) assuming a weakly adaptive adversary.[4]

The properties of these earlier works are summarized in Table 1, together with those of the algorithm we propose. Among these works, only [13] is deterministic and therefore tolerates a strongly adaptive adversary. It imposes, however, a strong constraint on n ($n > \max(4t, 2t^2-2t+2)$) and does not tolerate a majority of Byzantine processes, which the other algorithms do. Conversely, the algorithms of [3, 16, 27] all tolerate an arbitrary number of Byzantine processes, but contrary to the solution we present, they rely on randomization under a weakly adaptive adversary and do not exploit executions in which the number of Byzantine processes is less than the upper bound t. (They are not early stopping.)

3 Computing Model and Specification

3.1 System model

Process model. The system is composed of n synchronous sequential processes denoted $\Pi = \{p_1, ..., p_n\}$. Each process p_i has an identity; all the identities are different and known by all processes. To simplify, we assume that i is the identity of p_i.

Regarding failures, up to t processes can be Byzantine, where a Byzantine process is a process whose behavior does not follow the code specified by its algorithm [20, 24]. Let us notice that Byzantine processes can collude to fool the non-Byzantine processes (also called

[3] More precisely, this expected number of rounds can be broken down into a deterministic number of synchronous rounds followed by an expected number of asynchronous rounds. The exact breakdown depends, in turn, on the choice of shared random coin used in the algorithm.

[4] Although correct processes can deliver their message in about $2 \times n/(n-t)$ rounds in this optimized algorithm, they must continue to participate in the algorithm for about the same amount of time, leading to an overall execution time of circa $4 \times n/(n-t)$ rounds in good-cases.

correct processes). Let us also notice that, in this model, the premature stop (crash) of a process is a Byzantine failure. c denotes the number of processes that effectively behave correctly in an execution. Both c and n remain unknown to correct processes, but they are used to analyze the properties of our algorithm.

Network model. Processes communicate by exchanging messages through a reliable synchronous network, in which messages are delivered in the round in which they were sent.

Security model. As earlier works in this area [13, 16, 20, 24, 27], we assume a PKI (Public Key Infrastructure) that provides an ideal signature scheme. Processes can sign the messages they send, verify signatures, and forward content signed by other processes.

3.2 Byzantine Reliable Broadcast

Following [3, 16, 27], we consider a one-shot Byzantine-tolerant reliable broadcast (BRB for short) in which the sending process p_{sender} is known beforehand. The BRB abstraction provides two operations, brb_broadcast and brb_deliver. brb_broadcast(m) is invoked by the sending process p_{sender}. When this happens, we say that p_{sender} brb-broadcasts m. When a process p_i invokes brb_deliver(m) we say that p_i brb-delivers m. The BRB abstraction is specified by the following five properties.

- Safety:
  - BRB-Validity: If a correct process p_i brb-delivers a message m and p_{sender} is correct, then p_{sender} has brb-broadcast m.
  - BRB-No-duplication: A correct process p_i brb-delivers at most one message.
  - BRB-No-duplicity: No two different correct processes brb-deliver different messages.
- Liveliness:
  - BRB-Local-delivery: If p_{sender} is correct and brb-broadcasts a message, then at least one correct process p_j eventually brb-delivers some message.
  - BRB-Global-delivery: If a correct process p_i brb-delivers a message, then all correct processes brb-deliver a message.

4 A deterministic synchronous BRB algorithm

4.1 Underlying intuition

Signature chains. The original BRB algorithm of Lamport, Shostak, and Pease uses *signature chains* to propagate what each process knows of the system's state [20]. A signature chain (or chain for short) starts by a message m signed by the sending process, e.g. $(m, i_{\text{sender}}, \sigma_{p_{\text{sender}}})$, where i_{sender} is the identify of the sending process, and $\sigma_{p_{\text{sender}}}$ is a signature of (m, i_{sender}) with p_{sender}'s private key. Such a chain is of length 1, as it contains one signature. A chain of length ℓ is extended by appending the identity $i_{\ell+1}$ of a process $p_{i_{\ell+1}}$ not present in the chain, followed by $p_{i_{\ell+1}}$'s signature of the resulting sequence:

$$(m, i_{\text{sender}}, \sigma_{p_{\text{sender}}}, i_2, \sigma_{p_{i_2}}, .., i_\ell, \sigma_{p_{i_\ell}}, i_{\ell+1}, \sigma_{p_{i_{\ell+1}}}).$$

As in [14, 20], we use the compact notation $m : p_{\text{sender}} : p_{i_2} : .. : p_{i_{\ell+1}}$ to represent such a chain.

Valid chains. In Lamport, Shostak, and Pease's original algorithm [20], further formalized in [14], and algorithms based on the same idea [16], correct processes only accept *valid* signature chains, i.e., signature chains that are acyclic and whose length matches the current round. These conditions constrain the disruption power of Byzantine processes by limiting how long they can hide a message from correct processes. In [14, 20], a message is considered for delivery when backed by at least one chain containing $t+1$ signatures: the length of the chain $(t+1)$ ensures that Byzantine processes cannot reveal some message m to only a subset of correct processes, while hiding it from others, and thus guarantees that all correct processes use the same set of messages to decide which message should be delivered (using a deterministic choice function).

From chains to certificates. The protocol we propose generalizes this intuition in a simple, albeit non-trivial, way. Instead of single chains, our algorithm uses *sets of chains* forming a particular *pattern* to trigger delivery. We call these chain patterns *certificates*. We constrain how a certificate might be propagated to limit how long Byzantine processes can hide a valid certificate from correct processes. A given certificate for a message m has a "weight" representing how many processes are "backing" m. To back a message m, a process must have witnessed it at the latest by the end of round 2. The heavier a certificate, the more quickly a correct process can make a delivery decision in the absence of any contradictory information. This approach is beneficial when the initial sender is correct , allowing correct processes to terminate in this case in $\mathsf{max}(2, t+3-c)$ rounds[5].

4.2 Notations

We use the following notations:

- $m:p_{i_1}:p_{i_2}:\cdots:p_{i_\ell}$ is a chain of signatures (or *chain* for short) as in [14, 16, 20]. We say that the *length* of the chain is ℓ. A *valid* chain must start with p_{sender} (i.e. $p_{\text{sender}} = p_{i_1}$), only contain valid signatures, and be acyclic (a process' signature can only appear once in a given chain). As in [14], we assume a filter function removes any invalid chain from the reception queue of correct processes, so that correct processes only receive valid chains. In particular, correct processes will only accept chains of length R during round R. As a shortcut, we might therefore say that a process p_i *has signed a chain* π *in round* R to mean that p_i's signature is the Rth signature in π.
- π being a chain of signatures, $\mathsf{message}(\pi)$ denotes the message at the start of the chain. We therefore have $\mathsf{message}(m:p_{\text{sender}}:p_{i_2}:\cdots:p_{i_\ell}) = m$. By extension, if E is a set of chains, $\mathsf{message}(E)$ is the direct image of E by $\mathsf{message}()$.
- M being a set of messages, $\mathsf{choice}(M)$ deterministically returns one of the messages, i.e., the same message m is returned by all correct processes for the same input set M. The function $\mathsf{choice}()$ can be implemented in various ways: e.g., the message with the smallest value or smallest time-stamp. If M is empty, $\mathsf{choice}(M)$ returns $\perp$.
- $\gamma = (p_{i_{k,\gamma}})_{k\in[1..\ell]} \in \Pi^\ell$ being a sequence of ℓ processes, for simplicity, we will use the notation $:\gamma:$ as a shorthand for the fragment of signature chain $:p_{i_{1,\gamma}}:\cdots:p_{i_{\ell,\gamma}}:$. For instance, $m:p_{\text{sender}}:\gamma:p_i$ thus means $m:p_{\text{sender}}:p_{i_{1,\gamma}}:\cdots:p_{i_{\ell,\gamma}}:p_i$. We will similarly equate the sequence γ with its supporting set $\mathsf{set}(\gamma) = \{p_{i_{k,\gamma}}\}_{k\in[1..\ell]}$ when unambiguous. Thus $q \in \gamma$ means $q \in \{p_{i_{k,\gamma}}\}_{k\in[1..\ell]}$, $|\gamma| = |\{p_{i_{k,\gamma}}\}_{k\in[1..\ell]}| = \ell$, $X \cup \gamma = X \cup \{p_{i_{k,\gamma}}\}_{k\in[1..\ell]}$.

[5] Similarly to other works studying synchronous broadcast with a dishonest majority [3, 16, 27], the presented algorithm considers crashed processes as Byzantine, providing no guarantees for them. A simple change, however, which adds one extra round, can ensure that crashed processes that brb-delivers benefit from the BRB-No-duplicity and BRB-Global-delivery properties. See footnote 6.

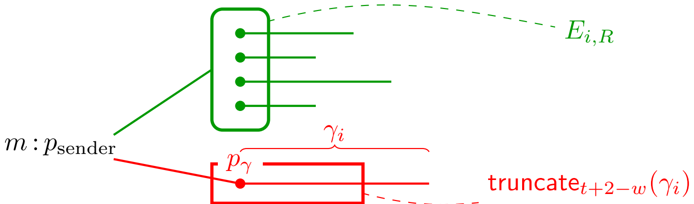


Figure 1 The pattern of signature chains forming a certificate of weight $w = 6$ (p_{sender}, p_γ and the processes of $E_{i,R}$) at round R for message m at p_i. The certificate must verify $\mathsf{set}(\mathsf{truncate}_{t+2-w}(\gamma_i)) \cap E_{i,R} = \emptyset$, which ensures its "conspicuity" (Lemma 4).

- $\gamma = (p_{i_{k,\gamma}})_{k\in[1..\ell]} \in \Pi^\ell$ being a sequence of ℓ processes, we note $\mathsf{truncate}_k(\gamma)$ the subsequence of γ that contains up to its first k elements $(p_{i_{k,\gamma}})_{k\in[1..\mathsf{min}(\ell,k)]}$. If $|\gamma| \le k$ in particular, $\mathsf{truncate}_k(\gamma) = \gamma$.

5 Description of the algorithm

5.1 Overview

Certificates and revealing chains. The algorithm revolves around the notion of *certificate*, which can be informally described as a set of signature chains for a given message m that fits a particular pattern. The *weight* of a certificate is defined as the number of processes whose signature appears within the first two positions of some chains of the certificate. These processes are said to be *backing* m in the certificate.

Just counting and propagating the round-2 signatures that correct processes observe is, however, not enough, as it does not prevent Byzantine processes from hiding part of a certificate from correct processes until the very last moment (round $t + 1$ in our case). The certificates we use therefore add an additional constrain that limits the disruption power of Byzantine processes: a certificate of weight w must contain a "*revealing chain*" $m : p_{\text{sender}} : \gamma_i$ (shown in red in Figure 1) whose makeup must "differ sufficiently" from the backing processes documented by the certificate. "Differ sufficiently" means that besides the processes in position 1 (p_{sender} in all cases) and position 2 (p_γ in Figure 1), the processes from position 3 until position $t + 3 - w$ of this revealing chain should not be backing processes.

This constraint limits what Byzantine processes can do when the sender is Byzantine and allows correct processes to use an early delivery condition that is safe both in good and bad cases. When p_{sender} is Byzantine (bad case), Byzantine processes may collude to forge competing certificates for different messages. When doing so, however, Byzantine processes can only use up to t signatures and must decide whether to invest these t signatures in the backing part of each certificate (thus increasing the certificate's weight) or in the revealing chain of the certificate (thus delaying the time at which the message of a forged certificate must be revealed to correct processes, but reducing the certificate's weight).

Certificate conspicuity. The position $t+3-w$ of the revealing chain enforces this constraint. The signatures from positions 3 to $t + 3 - w$ correspond to $(t + 3 - w) - 3 + 1 = t + 1 - w$ processes. Added to the w processes backing the certificate ($E_{i,R} \cup \{p_{\text{sender}}, p_\gamma\}$ in Figure 1), this represents $t + 1 - w + w = t + 1$ processes. These $t + 1$ processes must contain a correct process; therefore, Byzantine processes that seek to forge a certificate must include the signature of a correct process at the latest in round $t + 3 - w$. This correct process ensures

that the message of a forged certificate must be revealed to all correct processes at the latest by round $t+3-w$. We call the round $R_w = t+3-w$ the *conspicuity round* for weight w, and this property *Certificate Conspicuity.*

By contrapositive, certificate conspicuity allows correct processes to ascertain the *nonexistence* of a certificate of a given weight for a message. This ability to be sure that a given certificate does not exist, and the ability to propagate certificates that do, are the key ingredients that allow our algorithm to terminate (much) faster than other chain-based deterministic algorithms [14, 16, 20] in good cases, more precisely in $\mathsf{max}(2, t+3-c)$ rounds, where $c = n - f$ is the number of effective correct processes.

An example of certificate. Figure 1 shows a certificate of weight $w = 6$ for a message m observed by p_i at round R: each horizontal line represents a chain of signatures that starts with $m : p_{\text{sender}}$, the green and red dots represent processes that have witnessed $m : p_{\text{sender}}$ in round 2 (and are therefore backing m), and $m : p_{\text{sender}} : \gamma_i$ is the revealing chain, such that the process appearing from position 3 to $t+3-w$ ($t-3$ here) in $m : p_{\text{sender}} : \gamma_i$ (or equivalently from position 2 to $t+2-w$ in γ_i) do not appear in position 2 of any of the certificate's chains (equivalently the $t+2-w$ truncation of γ_i, noted $\mathsf{truncate}_{t+2-w}(\gamma_i)$, does not appear in $E_{i,R}$, the set of processes in position 2 in chains of the certificate other than $m : p_{\text{sender}} : \gamma_i$).

The certificate depicted in Figure 1 is of weight $w = 6$, as it proves that 6 distinct processes are backing m, i.e. they have signed a chain containing m in round 1 (for p_{sender}) or 2 (for the others). These processes are p_{sender}, p_γ (the first process in γ_i), and the four processes of $E_{i,R}$.

A special case: delivery in round 2. A special case occurs when the weight of a certificate reaches $w = t+1$. When this happens, any process p_i observing the certificate knows that either p_{sender}, p_γ, or one of the processes of $E_{i,R}$ is correct and, therefore, that all correct processes must have received a chain containing m by round 2. Conversely if p_i has not received any chain containing a message m' by round 2, p_i knows that a certificate of weight $t+1$ cannot possibly exist for m'. As a result, a correct process that observes a certificate a weight $t+1$ for m and is not aware of any other message $m' \neq m$ by round 2 can safely brb-deliver m, as no other message will be able to "beat" m with a heavier certificate, even if the sender p_{sender} is Byzantine.

Weak non-intersecting quorums. The reasoning for $w = t+1$ mirrors the mechanism of intersecting quorums used in asynchronous systems and requires a majority of correct processes (or $n > 2t$) to be guaranteed to occur when the sender is correct. The proposed certificate mechanism leverages the additional guarantees that a synchronous system brings to generalize this idea to weaker non-intersecting "quorums", whose ability to trigger a brb-delivery decision requires additional temporal information (waiting until the conspicuity round $R_w = t+3-w$).

5.2 Algorithms

In the pseudo-code of our algorithm, we use the operation $\mathsf{broadcast}(m)$ as a shorthand for **for all** $p_j \in \Pi$ **do** send m to p_j **end for**.

For readability, the pseudo-code for the sending process p_{sender} is presented separately in Algorithm 1. To brb-broadcast m, p_{sender} simply signs m and produces the signature chain $m : p_{\text{sender}}$ and broadcasts a protocol message $\textsc{msg}(\{m : p_{\text{sender}}\})$ containing this chain to all correct processes, before brb-delivering m locally.

Algorithm 1 brb-broadcast operation executed by p_{sender}.

```
1  In synchronous round R = 1 do
2  │  broadcast MSG({m : p_sender})
3  │  brb_deliver(m)
4  end round
```

Algorithm 2 Certificate-based Synchronous BRB code for $p_i \neq p_{\text{sender}}$.

```
1  Init:
2  │  view_i ← [∅, ..., ∅]  ▷Array of size t + 1 containing the chains observed by p_i in each round
3  │  ready_i ← false ; to_be_bcast_{i,1} ← ∅
4  end init
5  In each synchronous round R ∈ [1..t + 1] do
   │  Communication step
6  │  broadcast MSG(to_be_bcast_{i,R})                         ▷p_i receives its own broadcast
7  │  view_i[R] ← {π ∈ chains_{j,R}, such that MSG(chains_{j,R}) ∈ received_{i,R}}
8  │  to_be_bcast_{i,R+1} ← {π : p_i | π ∈ view_i[R] ∧ p_i ∉ π}
   │  Computation step
9  │  if ready_i then quit()
10 │  known_msgs_{i,R} ← message(⋃_{r∈[1..R]} view_i[r])
11 │  if R = t + 1 then
12 │  │  W_i ← {w ∈ ℕ | ∃m ∈ known_msgs_{i,t+1} : certificate_i(m, w)} ; if W_i = ∅ then quit()
13 │  │  wmax_i ← max (W_i)
14 │  │  candidate_msgs_i ← {m ∈ known_msgs_{i,t+1} | certificate_i(m, wmax_i)}
15 │  │  brb_deliver(choice(candidate_msgs_i))
16 │  elseif known_msgs_{i,R} = {m} ∧ certificate_i(m, t + 3 − R) then
17 │  │  brb_deliver(m) ; ready_i ← true
18 │  end if
19 end round
```

Algorithm 2 constitutes the core of the proposed BRB. It uses up to $t + 1$ synchronous rounds (lines 5-19). Each round is divided into a communication step (lines 6-8), during which processes broadcast and receive messages exchanged during the round, and a computation step (lines 9-18) during which they handle received messages and prepare the messages to be sent during the next round. $received_{i,R}$ represents the messages received by process p_i during round R. It is directly updated by the (synchronous) network layer.

R is a global variable containing the sequence number of the current round. $to_be_bcast_{i,R}$ contains the signature chains to be broadcast by p_i during round R. In the first round $p_i \neq p_{\text{sender}}$ broadcasts an empty protocol message $\textsc{msg}(\emptyset)$. p_i stores in the array $view_i[R]$ the signature chains it receives during round R (line 7). Chains that do not already contain p_i's signature are signed by p_i and stored for broadcasting in the next round (line 8).

Algorithm 3 Certificate function.

```
1 Function certificate_i(m, w) is
2     S_{i,R} ← { q ∈ Π | m : p_sender : q ∈ truncate_2 ( ⋃_{r′∈[2..t+1]} view_i[r′] ) }
3     return ∃r_i ∈ [2..t+1], ∃γ_i ∈ Π^{r_i−1} : { m : p_sender : γ_i ∈ view_i[r_i] ∧
                                                 |S_{i,R} \ truncate_{t+2−w}(γ_i)| ≥ w − 2
4 end function
```

p_i's behavior in the computation step depends on whether p_i has reached round $t+1$ or not. In earlier rounds, p_i used the conspicuity property of certificates to detect if a message m is backed by a certificate "heavy enough" that cannot be beaten by any other message $m' \neq m$ (condition at line 16). If this is the case, m is brb-delivered at line 17, and the flag $ready_i$ is toggled to stop the algorithm in the next round.[6] "Heavy enough" means that p_i should observe a certificate of weight at least $w_R = t+3-R$ for m. The value w_R is simply the weight whose conspicuity round turns out to be R, as $t+3-w_R = t+3-(t+3-R) = R$ (see Section 5.1). This implies that, by round R, all certificates of weight at least $w_R = t+3-R$ must have become conspicuous and allows p_i to make a safe brb-delivery.

If p_i reaches round $t+1$ without having delivered any message (line 11), it tallies all messages known to it and keeps only messages backed by a certificate with maximal weight. p_i uses a deterministic function choice to break any tie that may appear.

The code for the function $\mathsf{certificate}_i$ executed by p_i is shown in Algorithm 3. $\mathsf{certificate}_i$ first computes the set of all length-2 prefixes of signature chains known to p_i (set $S_{i,R}$ at line 2), and seeks to find a "revealing chain" $m : p_{\text{sender}} : \gamma_i$ known to p_i so that after removing the $t+2-w$ truncation[7] of γ_i from $S_{i,R}$, enough distinct processes[8] remain to ensure w processes have signed m by round 2 (see Figure 1 and Section 5.1.)

In terms of vocabulary, we say that p_i *observes a certificate* of weight w for a message m during round R if $\mathsf{certificate}_i(m,w) = \mathtt{true}$ during the computation step of round R at p_i (lines 9-18 of Algorithm 2). Note that because $view_i[R]$ is initially empty and is only modified once (during round R, line 2 of Algorithm 2), $\mathsf{certificate}_i(m,w)$ is stable (once $\mathtt{true}$ during some round, $\mathsf{certificate}_i(m,w)$ remains true in all subsequent rounds). For the same reasons, and by definition of $\mathsf{certificate}_i$ (Algorithm 3), if $\mathsf{certificate}_i(m,w) = \mathtt{true}$ for some weight w, then $\mathsf{certificate}_i(m,w') = \mathtt{true}$ for any smaller weight $w' \leq w$.

6 Proof of correctness

▶ **Theorem 1.** *Algorithm 2 implements a Synchronous Byzantine Reliable Broadcast object. If the initial sender p_{sender} is correct, correct processes brb-deliver in at most* $\max(2, t+3-c)$ *rounds, where c is the effective number of correct processes.*

[6] The extra round of communication induced by $ready_i$ is needed to ensure all correct processes observe the same certificate as p_i. However, by delivering as soon as the condition of line 16 is true, the algorithm does not ensure that crashed processes benefit from the BRB-No-duplicity and BRB-Global-delivery properties. These additional guarantees can be provided at the cost of one extra round by postponing the brb-delivery of m by one round from line 17 to line 9. See footnote 5.

[7] The $t+2-w$ truncation of γ_i equals the subchain of $m : p_{\text{sender}} : \gamma_i$ between positions 2 and $t+3-w$.

[8] $\mathsf{certificate}_i(m,w)$ uses the threshold $w-2$ at line 3 to take into account that p_{sender} and the first process of γ_i (called p_γ in Figure 1) are also backing m. See our discussion in Section 5.1.

▶ Remark 2. Note that if $n > 2t$, then because $c \geq n - t$, we have $c \geq t + 1$, and $\mathsf{max}(2, t + 3 - c) = 2$, all correct processes deliver in at most 2 rounds when the sender is correct.

▶ Remark 3. Algorithm 2 can easily be adapted to solve Byzantine Broadcast by modifying line 12 to brb-deliver some default value (e.g., $\perp$) when the set W_i is empty.

6.1 Preliminary lemmas

Theorem 1 hinges on two fundamental properties: *Certificate Conspicuity* (Lemma 4) and *Certificate Final Visibility* (Lemma 9). Lemmas 6-7 are used to prove Lemma 9.

As explained earlier, *Certificate Conspicuity* forces Byzantine processes that seek to hide a certificate of weight w for a message m to reveal at the latest by round $t + 3 - w$ (the "conspicuity round" of w) that m exists. *Certificate Final Visibility* ensures that when the initial sender p_{sender} is malicious, if a correct process reaches round $t + 1$, then this correct process observes all the certificates ever observed by other correct processes.

▶ **Lemma 4** (Certificate Conspicuity). *Let p_i and $p_j \neq p_i$ be correct processes, m a message, and $R \in [2..t + 1]$ a round. If p_i observes a certificate of weight at least $t + 3 - R$ for m at some point of its execution (i.e.* $\mathsf{certificate}_i(m, t + 3 - R) = \mathtt{true}$*) and p_j executes round R, then $m \in known_msgs_{j,R}$ at round R at p_j.*

Sketch of proof. (Detail in the appendix.) Let us note r_i and γ_i a round and a process sequence that render true the condition at line 3 in the definition of $\mathsf{certificate}_i$ for p_i (Alg. 3). The proof depends on whether r_i (the round in which p_i observes the revealing chain $m : p_{\text{sender}} : \gamma_i$, cf. Alg. 3) occurs before or after the round R, the round during which we seek to prove that all correct processes are aware of m. If $r_i < R$, because p_i forwards all chains it has not signed yet, all correct processes observe a chain containing m at the latest by round $r_i + 1 \leq R$. If $r_i \geq R$, the fact that only the first process of $\mathsf{truncate}_{t+2-w}(\gamma_i)$ can be backing m in p_i's certificate implies that $(t + 2 - w) + (w - 2) + 1 = t + 1$ processes (counting the $(t + 2 - w)$-prefix of γ_i, the remaining processes of $S_{i,R}$ not in the prefix, and p_{sender}) have signed a chain containing m during the first R rounds of the protocol. One of them must be correct, yielding the lemma. ◀

The following corollary from Lemma 4 states that if a correct process has not seen a message m by round $R \geq 2$, then no certificate of weight $\geq t + 3 - R$ will ever exist.

▶ **Corollary 5.** *Let p_i and $p_j \neq p_i$ be correct processes, m a message, and $R \in [2..t + 1]$ a round. If $m \notin known_msgs_{i,R}$ at round R at line 16 of Algorithm 2 at p_i, then for any $R' \geq R$,* $\mathsf{certificate}_j(m, t + 3 - R') = \mathtt{false}$ *during all of p_j's execution.*

Proof. Consider p_i and p_j two correct processes. If $m \notin known_msgs_{i,R}$ at round R of p_i, then by contrapositive of Lemma 4, then $\mathsf{certificate}_j(m, t + 3 - R) = \mathtt{false}$ during all of p_j's execution. Because of the inequality at line 3 of the definition $\mathsf{certificate}$ (Alg. 3), $\mathsf{certificate}_j(m, w)$ implies $\mathsf{certificate}_j(m, w')$ for any $w' \leq w$, and therefore $\mathsf{certificate}_j(m, t + 3 - R) = \mathtt{false}$ implies $\mathsf{certificate}_j(m, t + 3 - R') = \mathtt{false}$ for any $R' \geq R$. ◀

In the coming lemmas, we use the following quantity to prove the final visibility of certificates (Lemma 9), which is central to establishing the BRB-No-duplicity property.

$$\text{Let } T_{2,i}[R] \ \text{ denote } \mathsf{truncate}_2\left(\bigcup_{r' \in [2..R]} view_i[r']\right). \tag{1}$$

$T_{2,i}[R]$ contains all length-2 prefixes $m : p_{\text{sender}} : q$ observed by p_i by round R, i.e. p_i's knowledge during round R of the processes that have signed m by the end of round 2.

The following lemma states that all length-2 prefixes known by a correct process $p_i \neq p_{\text{sender}}$ at round R are known by all other correct processes by round $R+1$.

▶ **Lemma 6.** *Let p_i and $p_j \neq p_i$ be two correct processes, such that p_i executes the computation step (lines 6-8) of at least the $R \leq t$ first rounds, and p_j executes the communication step of at least the first $R+1$ rounds. Then we have $\forall R \in [1..t]$, $T_{2,i}[R] \subseteq T_{2,j}[R+1]$.*

Proof. Note that since p_i and p_j execute Algorithm 2, they are both different from p_{sender}. We prove the lemma by induction.

- Case $R = 1$: $\bigcup_{r' \in [2..1]} \mathit{view}_i[r'] = \emptyset$, and therefore $T_{2,i}[1] = \emptyset$, trivially proving the case.
- Induction case: Let us assume $T_{2,i}[R] \subseteq T_{2,j}[R+1]$ for some $R \in [1..t-1]$.

$$
\begin{aligned}
T_{2,i}[R+1] &= \mathsf{truncate}_2\left(\textstyle\bigcup_{r' \in [2..R+1]} \mathit{view}_i[r']\right), \\
&= \mathsf{truncate}_2\left(\textstyle\bigcup_{r' \in [2..R]} \mathit{view}_i[r'] \cup \mathit{view}_i[R+1]\right), \\
&= \mathsf{truncate}_2\left(\textstyle\bigcup_{r' \in [2..R]} \mathit{view}_i[r']\right) \cup \mathsf{truncate}_2\big(\mathit{view}_i[R+1]\big), \\
&= T_{2,i}[R] \cup \mathsf{truncate}_2\big(\mathit{view}_i[R+1]\big), \\
&\subseteq T_{2,j}[R+1] \cup \mathsf{truncate}_2\big(\mathit{view}_i[R+1]\big) && \text{by case assumption,} \\
&\subseteq T_{2,j}[R+2] \cup \mathsf{truncate}_2\big(\mathit{view}_i[R+1]\big) && \text{as } T_{2,j}[R+1] \subseteq T_{2,j}[R+2].
\end{aligned}
$$

We now need to show that $\mathsf{truncate}_2\big(\mathit{view}_i[R+1]\big) \subseteq T_{2,j}[R+2]$ to complete the proof. Consider $m : p_{\text{sender}} : \gamma \in \mathit{view}_i[R+1]$, with $\gamma \in \Pi^R$. By assumption, $p_i \neq p_{\text{sender}}$, we must therefore distinguish two cases depending whether p_i appears in γ or not.

- Case 1: If $p_i \in \gamma$, p_i has signed a chain $m : p_{\text{sender}} : \gamma'$ at line 8 of Alg. 2 during a round $R' < R+1$ (where $\gamma' : p_i$ is a prefix of γ), and p_i has broadcast the chain $m : p_{\text{sender}} : \gamma' : p_i$ to all processes (since p_i is correct) at line 6 during the communication step of the following round $R'+1 \leq R+1$. Therefore $m : p_{\text{sender}} : \gamma' : p_i \in \mathit{view}_j[R'+1]$, which implies

$$\mathsf{truncate}_2(m : p_{\text{sender}} : \gamma) = \mathsf{truncate}_2(m : p_{\text{sender}} : \gamma' : p_i) \in \mathsf{truncate}_2(\mathit{view}_j[R'+1]) \subseteq T_{2,j}[R+2].$$

- Case 2: If $p_i \notin \gamma$, p_i signs $m : p_{\text{sender}} : \gamma$ during round $R+1$ and as above broadcasts $m : p_{\text{sender}} : \gamma : p_i$ at round $R+2$ to all processes. (By construction, the fact that p_i executes the computation step of round $R+1 \leq t$ implies that it executes the communication step of round $R+2$.) This similarly implies

$$\mathsf{truncate}_2(m : p_{\text{sender}} : \gamma) = \mathsf{truncate}_2(m : p_{\text{sender}} : \gamma : p_i) \in \mathsf{truncate}_2(\mathit{view}_j[R+2]) \subseteq T_{2,j}[R+2].$$

These two cases show that $\mathsf{truncate}_2\big(\mathit{view}_i[R+1]\big) \subseteq T_{2,j}[R+2]$, which concludes the proof of the lemma. ◀

The following lemma shows that if p_{sender} is Byzantine then all correct processes agree on the length-2 prefixes they have observed by round $t+1$.

▶ **Lemma 7.** *Let p_{sender} be Byzantine, and p_i and p_j be two correct processes that execute the communication step of round $t+1$, then $T_{2,i}[t+1] = T_{2,j}[t+1]$.*

Sketch of proof. (Detail in the appendix.) The proof uses the fact that the length-2 prefixes that p_i receives in round $t+1$ have been propagated by $t+1$ processes. One of these processes must be correct, and because p_{sender} is Byzantine, it must be a process that signed the chain at the earliest in round 2, implying that the length-2 prefix is also known to p_j. This observation, together with Lemma 6 yields the proof. ◀

▶ **Corollary 8.** *Let p_{sender} be Byzantine, p_i and p_j be two correct processes, such that p_i executes the computation step of at least the first $r \in [1..t+1]$ rounds, and p_j executes the communication step of all $t+1$ rounds. Then we have $T_{2,i}[r] \subseteq T_{2,j}[t+1]$.*

Proof. The proof follows either from Lemma 7 or 6, depending on whether $r = t+1$ or not.

- If $r = t+1$, the corollary follows trivially from Lemma 7.
- If $r < t+1$, this follows from Lemma 6, and observing that $T_{2,j}[r+1] \subseteq T_{2,j}[t+1]$. ◀

The following lemma states that if p_{sender} behaves maliciously (for instance, by sending different messages in round 1), if a certificate of weight w exists for a message m (meaning that it is observed at some point by some correct process), then all correct processes that reach round $t+1$ observe a certificate of weight w for m by round $t+1$, a property we have dubbed *Final Certificate Visibility.*

▶ **Lemma 9** (Certificate Final Visibility). *Let p_{sender} be Byzantine, and p_i and p_j be correct processes such that p_i observes $\mathsf{certificate}_i(m, w) = \mathtt{true}$ at some round $R \in [1..t+1]$, and p_j executes the communication step of round $t+1$. Then p_j observes $\mathsf{certificate}_j(m, w) = \mathtt{true}$ at round $t+1$.*

Proof. Assume a correct process p_i observes $\mathsf{certificate}_i(m, w) = \mathtt{true}$ at some round R. Consider p_j another correct process that reaches round $t+1$. Without loss of generality, assume $p_j \neq p_i$ (as the case $p_i = p_j$ is trivial).

In the following, r_i and γ_i denote a round and a process sequence that render true the condition at line 3 for p_i in the definition of $\mathsf{certificate}_i$ (Algorithm 3).

Let $E_{i,R}$ be the value of $S_{i,R} \setminus \mathsf{truncate}_{t+2-w}(\gamma_i)$ at line 3 of Alg. 3 at p_i in round R

$$E_{i,R} = \left\{ q \in \Pi \setminus \mathsf{truncate}_{t+2-w}(\gamma_i) \;\middle|\; m : p_{\text{sender}} : q \in \mathsf{truncate}_2 \left(\textstyle\bigcup_{r' \in [2..t+1]} view_i[r'] \right) \right\}. \quad (2)$$

By lemma assumption, $|E_{i,R}| \geq w - 2$. Furthermore, as $view_i[r']$ is initially empty and only updated during round r', during round R, $\forall r' > R : view_i[r'] = \emptyset$. At round R, we therefore have $\mathsf{truncate}_2 \left(\bigcup_{r' \in [2..t+1]} view_i[r'] \right) = \mathsf{truncate}_2 \left(\bigcup_{r' \in [2..R]} view_i[r'] \right) = T_{2,i}[R]$. We can therefore rewrite (2) into

$$E_{i,R} = \{ q \in \Pi \setminus \mathsf{truncate}_{t+2-w}(\gamma_i) \mid m : p_{\text{sender}} : q \in T_{2,i}[R] \}. \quad (3)$$

The rest of the proof distinguishes two cases, depending on whether $p_j \in \gamma_i$ or not, with the case $p_j \notin \gamma_i$ leading to more sub-cases.

- Case $p_j \in \gamma_i$: As p_j is correct, $p_j \in \gamma_i$ implies p_j has signed and therefore received a chain $m : p_{\text{sender}} : \gamma_j$ at line 8 of Alg. 2 during some round $r_j < r_i$, where γ_j is a prefix of γ_i. As a result of this broadcast, at all rounds higher or equal to r_j, we have

  $$m : p_{\text{sender}} : \gamma_j \in view_j[r_j]. \quad (4)$$

  As γ_j is a prefix of γ_i, $\mathsf{truncate}_{t+2-w}(\gamma_j)$ is also a prefix of $\mathsf{truncate}_{t+2-w}(\gamma_i)$, which implies using (3)

  $$\begin{aligned} E_{i,R} &\subseteq \{ q \in \Pi \setminus \mathsf{truncate}_{t+2-w}(\gamma_j) \mid m : p_{\text{sender}} : q \in T_{2,i}[R] \}, \\ &\subseteq \{ q \in \Pi \setminus \mathsf{truncate}_{t+2-w}(\gamma_j) \mid m : p_{\text{sender}} : q \in T_{2,j}[t+1] \} \qquad \text{using Corollary 8.} \end{aligned}$$

  As $|E_{i,R}| \geq w - 2$ (since $\mathsf{certificate}_i(m, w) = \mathtt{true}$ at round R by lemma assumption), this last inclusion yields

  $$|\{ q \in \Pi \setminus \mathsf{truncate}_{t+2-w}(\gamma_j) \mid m : p_{\text{sender}} : q \in T_{2,j}[t+1] \}| \geq w - 2. \quad (5)$$

  Equations (4) and (5) render true line 3 of $\mathsf{certificate}_j(m, w)$ (Alg. 3) at round $t+1$, proving the lemma.

- Case $p_j \notin \gamma_i$: If p_j's signature is not in γ_i the approach to find r_j and γ_j that fulfill line 3 of Alg. 3 for p_j depends on the value of r_i and on whether p_i appears in the set $E_{i,R}$ of length-2 prefixes.
  Line 3 of Alg. 3 implies $view_i[r_i] \neq \emptyset$. As $view_i[r_i]$ is initially empty and only updated in the communication step of round r_i, this implies $R \geq r_i$.
  - Case $r_i = t+1$: $R \geq r_i$ yields $R \geq t+1$. As $R \leq t+1$ by lemma assumption, we conclude $R = t+1$. Since $p_{\text{sender}} \notin \gamma_i$ due to the acyclic nature of signature chains accepted by correct processes, $|\{p_{\text{sender}}\} \cup \gamma_i| = |\{p_{\text{sender}}\}| + |\gamma_i| = 1 + r_i - 1 = t+1$ (by case assumption). $\{p_{\text{sender}}\} \cup \gamma_i$ therefore contains at least one correct process p_k. As p_{sender} is Byzantine (by lemma assumption), $p_k \in \gamma_i$. this means that p_k has broadcast to all processes a chain $m : p_{\text{sender}} : \gamma_j$ during some round $r_j \leq t+1$, where γ_j is a prefix of γ_i. p_j has received this chain, and we therefore have

$$m : p_{\text{sender}} : \gamma_j \in view_j[r_j]. \tag{6}$$

    Furthermore, since γ_j is a prefix of γ_i, we have

$$\begin{aligned} E_{t+1,i} &\subseteq \{q \in \Pi \setminus \mathsf{truncate}_{t+2-w}(\gamma_j) \mid m : p_{\text{sender}} : q \in T_{2,i}[t+1]\}\,, \\ &\subseteq \{q \in \Pi \setminus \mathsf{truncate}_{t+2-w}(\gamma_j) \mid m : p_{\text{sender}} : q \in T_{2,j}[t+1]\}\,, \quad \text{using Lemma 7.} \end{aligned}$$

    As above, $|E_{t+1,i}| \geq w-2$ (since $R = t+1$, see above) implies that this last set contains at least $w-2$ elements. This fact with (6) renders true line 3 of $\mathsf{certificate}_j(m, w)$ (Alg. 3) at round $t+1$, proving the lemma.
  - Case $r_i < t+1$:
    * Sub-case $p_i \notin E_{i,R}$ or $|\gamma_i| \geq t+2-w$: Line 3 of Alg. 3 implies $m : p_{\text{sender}} : \gamma_i \in view_i[r_i]$. If $p_i \notin \gamma_i$, p_i signs the chain $m : p_{\text{sender}} : \gamma_i$ (line 8, Alg. 2) and broadcasts to all processes (since p_i is correct) the chain $m : p_{\text{sender}} : \gamma_i : p_i$ during the communication step of round $r_i + 1$. If $p_i \in \gamma_i$, this means p_i has already performed these two steps (signing and broadcasting) at some earlier round.
      In both cases, p_j receives a chain $m : p_{\text{sender}} : \gamma_i' : p_i$ during some round $r_j \leq r_i + 1$. By choosing $\gamma_j = \gamma_i' : p_i$, we therefore have

$$m : p_{\text{sender}} : \gamma_j \in view_j[r_j]. \tag{7}$$

      If $p_i \in \gamma_i$, then γ_j is a prefix of γ_i, $\mathsf{truncate}_{t+2-w}(\gamma_j) \subseteq \mathsf{truncate}_{t+2-w}(\gamma_i)$ and

$$E_{i,R} \subseteq \{q \in \Pi \setminus \mathsf{truncate}_{t+2-w}(\gamma_j) \mid m : p_{\text{sender}} : q \in T_{2,i}[R]\}\,. \tag{8}$$

      If $p_i \notin \gamma_i$ and $|\gamma_i| \geq t+2-w$, then $\gamma_i' = \gamma_i$, $\mathsf{truncate}_{t+2-w}(\gamma_j) = \mathsf{truncate}_{t+2-w}(\gamma_i)$ and (8) still holds. Finally if $p_i \notin \gamma_i$ and $|\gamma_i| < t+2-w$, then $\gamma_i' = \gamma_i$, $\mathsf{truncate}_{t+2-w}(\gamma_j)$ contains an extra terminal p_i compared to $\mathsf{truncate}_{t+2-w}(\gamma_i)$. However, by case assumption $|\gamma_i| < t+2-w$ implies $p_i \notin E_{i,R}$, and therefore (8) continues to hold. Using Corollary 8 on (8) we therefore have independently of γ_i

$$E_{i,R} \subseteq \{q \in \Pi \setminus \mathsf{truncate}_{t+2-w}(\gamma_j) \mid m : p_{\text{sender}} : q \in T_{2,j}[t+1]\}\,.$$

      As above, using $|E_{i,R}| \geq w-2$ on the above inclusion yields

$$|\{q \in \Pi \setminus \mathsf{truncate}_{t+2-w}(\gamma_j) \mid m : p_{\text{sender}} : q \in T_{2,j}[t+1]\}| \geq w-2. \tag{9}$$

      Equations (7) and (9) render true line 3 of $\mathsf{certificate}_j(m, w)$ (Alg. 3) at round $t+1$ (since $r_j = r_i + 1 \leq t+1$ by case assumption), proving the lemma.

* Sub-case $p_i \in E_{i,R}$ and $|\gamma_i| < t+2-w$: $p_i \in E_{i,R}$ means that p_i has signed $m : p_{\text{sender}}$. Since p_i is correct, it has therefore broadcast to all processes $m : p_{\text{sender}} : p_i$ during round 2. As a result, from round 2 onward

$$m : p_{\text{sender}} : p_i \in \mathit{view}_j[2]. \tag{10}$$

By definition $r_i \geq 2$ (line 3 of Alg. 3), and therefore $|\gamma_i| = r_i - 1 \geq 1$, implying γ_i is non-empty (and $t + 2 - w > 1$ by sub-case assumption). Consider p_γ the first process in γ_i. As $|\gamma_i| < t + 2 - w$ (sub-case assumption), $p_\gamma \in \mathsf{truncate}_{t+2-w}(\gamma_i)$, and therefore $p_\gamma \notin E_{i,R}$, by definition of $E_{i,R}$ (Equations 2 and 3).
We construct a set of length-2 prefixes for p_j by removing p_i and adding p_γ from/to $E_{i,R}$. More precisely, we have

$$\begin{aligned}
&\big(E_{i,R} \setminus \{p_i\}\big) \cup \{p_\gamma\} \\
&\quad = \big(\{q \in \Pi \setminus \mathsf{truncate}_{t+2-w}(\gamma_i) \mid m : p_{\text{sender}} : q \in T_{2,i}[R]\} \setminus \{p_i\} \big) \cup \{p_\gamma\}, \\
&\quad = \big(\{q \in \Pi \setminus \{p_i\} \mid m : p_{\text{sender}} : q \in T_{2,i}[R]\} \setminus \mathsf{truncate}_{t+2-w}(\gamma_i) \big) \cup \{p_\gamma\}, \\
&\quad \subseteq \{q \in \Pi \setminus \{p_i\} \mid m : p_{\text{sender}} : q \in T_{2,i}[R]\} \cup \{p_\gamma\}.
\end{aligned} \tag{11}$$

Moreover as $m : p_{\text{sender}} : \gamma_i \in \mathit{view}_i[r_i]$ (line 3 of Alg. 3), and $R \geq r_i$, $m : p_{\text{sender}} : p_\gamma = \mathsf{truncate}_2(p_{\text{sender}} : \gamma_i) \in T_{2,i}[R]$. Because $p_\gamma \neq p_i$ (since $p_i \in E_{i,R}$ and $p_\gamma \notin E_{i,R}$), this last statement implies $p_\gamma \in \{q \in \Pi \setminus \{p_i\} \mid m : p_{\text{sender}} : q \in T_{2,i}[R]\}$, which with (11) yields

$$\begin{aligned}
\big(E_{i,R} \setminus \{p_i\}\big) \cup \{p_\gamma\} &\subseteq \{q \in \Pi \setminus \{p_i\} \mid m : p_{\text{sender}} : q \in T_{2,i}[R]\}, \\
&\subseteq \{q \in \Pi \setminus \{p_i\} \mid m : p_{\text{sender}} : q \in T_{2,j}[t+1]\} \quad \text{using Corollary 8.}
\end{aligned}$$

As $p_i \in E_{i,R}$ by case assumption, and $p_\gamma \notin E_{i,R}$, $|\big(E_{i,R} \setminus \{p_i\}\big) \cup \{p_\gamma\}| = |E_{i,R}| - 1 + 1 = |E_{i,R}|$. As $|E_{i,R}| \geq w - 2$ (since $\mathsf{certificate}_i(m, w) = \mathtt{true}$ at round R by lemma assumption), the above inclusion leads to

$$\big|\{q \in \Pi \setminus \{p_i\} \mid m : p_{\text{sender}} : q \in T_{2,j}[t+1]\}\big| \geq \big|\big(E_{i,R} \setminus \{p_i\}\big) \cup \{p_\gamma\}\big| = |E_{i,R}| \geq w - 2. \tag{12}$$

By choosing $\gamma_j = p_i$ and $r_j = 2$, (10) and (12) render true line 3 of $\mathsf{certificate}_j(m, w)$ (Alg. 3) at round $t + 1$ (since $t \geq 1$, and $t + 1 \geq 2$), proving the lemma. ◀

6.2 Proof of Theorem 1

Sketch of proof. (Detail in the appendix.)

- BRB-Validity follows from the use of (secure) signatures and the fact that correct processes only accept valid signature chains.
- BRB-No-duplication is ensured by construction of the algorithm, and BRB-Local-delivery from the code executed by p_{sender} when it is correct.
- BRB-No-duplicity follows from BRB-Validity when p_{sender} is correct. When p_{sender} is Byzantine, the no-duplicity follows from the Conspicuity of the certificates (Lemma 4) and from their Final Visibility (Lemma 9).
  Certificate conspicuity ensures that if two processes p_i and p_j brb-deliver a message before round $t + 1$, the process with the "weaker" certificate (say p_j) must be aware of p_i's message when it brb-delivers its own message, and therefore must brb-deliver the same message as p_i, due to the condition at line 17.

If p_i brb-delivers before round $t+1$ and p_j during round $t+1$, then the Final Visibility of certificates guarantees that p_j will observe p_i's certificate at line 12. Corollary 5 implies that p_i's message is guaranteed to have the "heaviest" certificate, ensuring agreement. Finally, if both p_i and p_j deliver in round $t+1$, Final Visibility guarantees they see the same set of messages and certificates and that they brb-deliver the same message.

- The good-case latency of the algorithm, $\mathsf{max}(2, t+3-c)$ rounds, follow from the observation that all correct processes observe a certificate of weight c by the end of round 2 when the initial sender, p_{sender}, is correct. As no other message exists in the system, the condition at line 16 ensures that all correct processes have delivered p_{sender}'s message at the latest either by the end of round 2 or by the end of round $t+3-c$, whichever occurs first.
- BRB-Global-delivery follows from the above reasoning when p_{sender} is correct. When p_{sender} is Byzantine, the property follows from Certificate Finality. ◀

7 Conclusion

Considering n-process synchronous distributed systems where up to $t < n$ processes can be Byzantine, this paper explored the good-case latency of deterministic Byzantine reliable broadcast (BRB) algorithms, the time taken by correct processes to deliver a message when the initial sender is correct.

In contrast to their randomized counterparts, no deterministic BRB algorithm was known that exhibited a good-case latency better than $t+1$ (the worst-case bound) under a majority of Byzantine processes. This article has proposed a novel deterministic synchronous BRB algorithm that substantially improves on this earlier bound and provides a good case latency of $\mathsf{max}(2, t+3-c)$ rounds, where t is the upper bound on the number of Byzantine processes, and c the number of effectively correct processes in the considered run.

The algorithm that has been presented extends the "signature chain mechanism" first proposed four decades ago and allows correct processes to brb-deliver much earlier when the context is favorable. In particular, when the sender is correct, and there are enough effectively correct processes ($c > t$), our algorithm delivers in 2 rounds, thus outperforming all known dishonest-majority BRB algorithms (whether deterministic or randomized).

Several crucial open questions remain, in particular, whether the upper bound of $t+3-c$ rounds can be further improved (for instance, using techniques employed in sub-linear randomized algorithms [4, 27]). In terms of lower bounds, one might ask whether the lower bound of $\lceil n/(n-t) \rceil - 1$ shown in [4] can be refined to include the effective number of correct processes c, and whether this same lower bound can be strengthened in the deterministic case, for instance considering the fact that Byzantine Agreement cannot be solved in a (worst-case) sub-linear communication complexity using algorithms that tolerate a strongly adaptive adversary (which include deterministic algorithms) [1].

References

1 Ittai Abraham, T-H. Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Communication complexity of byzantine agreement, revisited. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 317–326, New York, NY, USA, 2019. doi:10.1145/3293611.3331629.

2 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In *IEEE Symposium on Security and Privacy (S&P)*, pages 106–118, 2020.

3 Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of byzantine broadcast: A complete categorization. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 331–341, 2021.

4 Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of byzantine broadcast: A complete categorization. In *arXiv:2102.07240v2*, pages 1–38, 2021.

5 Hagit Attiya and Jennifer L. Welch. *Distributed computing - fundamentals, simulations, and advanced topics (2. ed.)*. Wiley series on parallel and distributed computing. Wiley, 2004.

6 Alex Auvolat, Davide Frey, Michel Raynal, and François Taïani. Money transfer made simple: a specification, a generic algorithm and its proof. *Bulletin of EATCS*, 132, 2020.

7 Alex Auvolat, Davide Frey, Michel Raynal, and François Taïani. Byzantine-tolerant causal broadcast. *Theoretical Computer Science*, 885:55–68, 2021.

8 Mathieu Baudet, George Danezis, and Alberto Sonnino. Fastpay: High-performance byzantine fault tolerant settlement. In *ACM Advances in Financial Technologies*, pages 163–177, 2020.

9 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information & Computation*, 75(2):130–143, 1987.

10 Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

11 Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theoretical Computer Science*, 777:155–183, 2019.

12 Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *Dependable Systems and Networks (DSN)*, pages 26–38. IEEE, 2020.

13 Danny Dolev, Ruediger Reischuk, and H Raymond Strong. Early stopping in byzantine agreement. *Journal of the ACM*, 37(4):720–741, 1990.

14 Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

15 Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

16 Matthias Fitzi and Jesper Buus Nielsen. On the number of synchronous rounds sufficient for authenticated byzantine agreement. In *International Symposium on Distributed Computing (DISC)*, pages 449–463. Springer, 2009.

17 Rachid Guerraoui, Nikola Knezevic, Vivien Quéma, and Marko Vukolic. The next 700 BFT protocols. In *EuroSys*, pages 363–376. ACM, 2010.

18 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. *Distributed Computing*, 35(1):1–15, 2022.

19 Damien Imbs and Michel Raynal. Trading off t-resilience for efficiency in asynchronous byzantine reliable broadcast. *Parallel Processing Letters*, 26(4):1650017:1–1650017:8, 2016.

20 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

21 Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable random functions. In *40th IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 120–130, 1999.

22 Achour Mostéfaoui, Moumen Hamouma, and Michel Raynal. Signature-free asynchronous byzantine consensus with $t < n/3$ and $o(n^2)$ messages. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 2–9. ACM, 2014.

23 Kartik Nayak, Ling Ren, Elaine Shi, Nitin H. Vaidya, and Zhuolun Xiang. Improved extension protocols for byzantine broadcast and agreement. In *International Symposium on Distributed Computing (DISC)*, volume 179 of *LIPIcs*, pages 28:1–28:17, 2020.

24 Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

25 Michel Raynal. *Fault-Tolerant Message-Passing Distributed Systems – An Algorithmic Approach*. Springer, 2018.

26 Jun Wan, Hanshen Xiao, Srinivas Devadas, and Elaine Shi. Round-efficient byzantine broadcast under strongly adaptive and majority corruptions. In *18th Theory of Cryptography Conference (TCC)*, LNCS 12550, pages 412–456. Springer, 2020.

27 Jun Wan, Hanshen Xiao, Elaine Shi, and Srinivas Devadas. Expected constant round byzantine broadcast under dishonest majority. In *18th Theory of Cryptography Conference (TCC)*, LNCS 12550, pages 381–411. Springer, 2020.

A Appendices

A.1 Proofs of preliminary lemmas

▶ **Lemma 4** (Certificate Conspicuity). *Let p_i and $p_j \neq p_i$ be correct processes, m a message, and $R \in [2..t+1]$ a round. If p_i observes a certificate of weight at least $t+3-R$ for m at some point of its execution (i.e.* $\mathsf{certificate}_i(m, t+3-R) = \mathtt{true}$*) and p_j executes round R, then $m \in known_msgs_{j,R}$ at round R at p_j.*

Proof. Assume a correct process p_i observes $\mathsf{certificate}_i(m, t+3-R) = \mathtt{true}$. In the following, r_i and γ_i denote a round and a process sequence that render true the condition at line 3 in the definition of $\mathsf{certificate}_i$ (Algorithm 3).

The remainder of the proof distinguishes two cases, depending on whether $r_i < R$ or not.

- Case $r_i < R$: $m\!:\!p_{\text{sender}}\!:\!\gamma_i \in view_i[r_i]$ implies that p_i receives $m\!:\!p_{\text{sender}}\!:\!\gamma_i$ during the communication step of round $r_i < R$ (line 7 of Algorithm 2). If $p_i \notin \gamma_i$, p_i signs the chain (line 8, Alg. 2), and broadcast it during the communication step of round $r_i + 1 \leq R$ (line 6 of the same algorithm). If $p_i \in \gamma_i$, p_i has signed a chain $m\!:\!p_{\text{sender}}\!:\!\gamma'$ earlier, and broadcast the result before or during round r_i.
  In both cases, this means all other correct processes receive some chain $m\!:\!p_{\text{sender}}\!:\!\gamma_i'\!:\!p_i$ either during or before round R, and therefore that $m \in known_msgs_{j,R}$ at round R for all correct processes p_j that execute round R.
- Case $r_i \geq R$: Let us note $\gamma_{R-1} = \mathsf{truncate}_{R-1}(\gamma_i)$. $m\!:\!p_{\text{sender}}\!:\!\gamma_i \in view_i[r_i]$ at line 3 means that $|\gamma_i| = r_i - 1$. Since by case assumption $r_i \geq R$, $|\gamma_i| \geq R-1$, and therefore $|\gamma_{R-1}| = |\mathsf{truncate}_{R-1}(\gamma_i)| = R-1$.
  Let E denote $\left\{ q \in \Pi \setminus \gamma_{R-1} \;\middle|\; m\!:\!p_{\text{sender}}\!:\!q \in \mathsf{truncate}_2\left(\bigcup_{r' \in [1..r_i]} view_i[r']\right) \right\}$.
  The condition of line 3 implies $|E| \geq t+1-R$. By construction of E, $\gamma_{R-1} \cap E = \emptyset$. Similarly, because correct processes only accept acyclic signature chains, $p_{\text{sender}} \notin E$. For the same reason line 3 of Algorithm 3 implies $p_{\text{sender}} \notin \gamma_{R-1} \subseteq \gamma_i$. E, γ_{R-1}, and $\{p_{\text{sender}}\}$ are therefore pair-wise disjoint. We therefore have $|E \cup \gamma_{R-1} \cup \{p_{\text{sender}}\}| = |E| + |\gamma_{R-1}| + 1 \geq (t+1-R) + (R-1) + 1 = t+1$. $E \cup \gamma_{R-1} \cup \{p_{\text{sender}}\}$ therefore contains at least one correct process, p_k.
  - If $p_k = p_{\text{sender}}$, the sender is correct, and all correct processes observe message m during round 1.
  - If $p_k \in E$, p_k has signed a chain of length 2 with message m and has broadcast this chain to all processes during round 2.
  - Finally, if $p_k \in \gamma_{R-1}$, the facts that $\gamma_{R-1} = \mathsf{truncate}_{R-1}(\gamma_i)$ and $m\!:\!p_{\text{sender}}\!:\!\gamma_i \in view_i[r_i]$ at line 3 of Alg. 3 imply that p_k has signed a chain with message m and has broadcast this chain to all processes during or before round $R \leq r_i$.

  All three cases imply that all correct processes that have not stopped earlier have observed message m at the latest by the end of the communication step of round R, i.e., that $m \in known_msgs_{j,R}$ at line 16 of Algorithm 2. ◀

▶ **Lemma 7.** *Let p_{sender} be Byzantine, and p_i and p_j be two correct processes that execute the communication step of round $t+1$, then $T_{2,i}[t+1] = T_{2,j}[t+1]$.*

Proof. By definition

$$\begin{aligned} T_{2,i}[t+1] &= \mathsf{truncate}_2\left(\bigcup_{r' \in [2..t+1]} view_i[r']\right) \\ &= \mathsf{truncate}_2\left(\bigcup_{r' \in [2..t]} view_i[r'] \cup view_i[t+1]\right), \\ &= T_{2,i}[t] \cup \mathsf{truncate}_2\big(view_i[t+1]\big). \end{aligned}$$

Applying Lemma 6 we have $T_{2,i}[t] \subseteq T_{2,j}[t+1]$, which with the previous equality yields

$$T_{2,i}[t+1] \subseteq T_{2,j}[t+1] \cup \mathsf{truncate}_2\big(view_i[t+1]\big). \tag{13}$$

We now prove that $\mathsf{truncate}_2\big(view_i[t+1]\big) \subseteq T_{2,j}[t+1]$. Consider $m : p_{\text{sender}} : \gamma \in view_i[t+1]$. As p_i is correct, it only accepts acyclic signature chains, and $p_{\text{sender}} \notin \gamma$. This implies $|p_{\text{sender}} \cup \gamma| = |p_{\text{sender}}| + |\gamma| = 1 + t$. $\{p_{\text{sender}}\} \cup \gamma$ therefore contains at least one correct process, p_k. As p_{sender} is Byzantine by lemma assumption, $p_k \in \gamma$, and p_k therefore has signed a chain $m : p_{\text{sender}} : \gamma'$ at line 8 of Alg. 2 before or during round t, where $\gamma' : p_k$ is a prefix of γ. As a result, p_k has broadcast the resulting chain $m : p_{\text{sender}} : \gamma' : p_k$ to all other processes during the following round $R' \leq t+1$. This implies $m : p_{\text{sender}} : \gamma' : p_k \in view_j[R']$, and hence

$$\mathsf{truncate}_2(m : p_{\text{sender}} : \gamma) = \mathsf{truncate}_2(m : p_{\text{sender}} : \gamma' : p_k) \in \mathsf{truncate}_2(view_j[R']) \subseteq T_{2,j}[t+1].$$

This last equation shows that $\mathsf{truncate}_2(view_i[t+1]) \subseteq T_{2,j}[t+1]$, which injected in (13) yields $T_{2,i}[t+1] \subseteq T_{2,j}[t+1]$. By inverting p_i and p_j, by the same reasoning we obtain $T_{2,j}[t+1] \subseteq T_{2,i}[t+1]$, which concludes the Lemma's proof. ◀

A.2 Proofs of Theorem 1

The proof of Theorem 1 follows from Lemmas 10-15, which follow.

▶ **Lemma 10.** *Algorithm 2 verifies the* BRB-Validity *Property.*

Proof. Consider p_i a correct process.

- If $p_i = p_{\text{sender}}$, the brb-delivery of a message m at line 3 of Algorithm 1 trivially implies that p_i has executed Algorithm 1, and hence has brb-broadcast m.
- If $p_i \neq p_{\text{sender}}$, p_i may brb-deliver a message m either at lines 17 or 15 of Algorithm 2. In both cases, m belongs to some $known_msgs_{i,R}$ variable computed at line 10, and must therefore appear in a signature chain of the form $m : p_{i_1} : \cdots : p_{i_\ell}$ received by p_i at line 7. As p_i is correct, it only accepts and processes valid chains of signatures by assumption, in which m is first signed by p_{sender} (i.e. $p_{i_1} = p_{\text{sender}}$). Since p_{sender} is correct, and we have assumed signatures to be secure, for m to be signed by p_{sender}, p_{sender} must have executed line 2 of Algorithm 1, and must therefore have brb-broadcast m. ◀

▶ **Lemma 11.** *Algorithm 2 verifies the* BRB-No-duplication *Property.*

Proof. Trivially, this is because once a correct process executes a brb_deliver operation (either at line 3 of Algorithm 1, or lines 17 or 15 of Algorithm 2), it terminates its execution, either immediately or at line 9 in the next round, without invoking brb_deliver. ◀

▶ **Lemma 12.** *Algorithm 2 verifies the* BRB-Local-delivery *property.*

Proof. The property trivially follows from the code executed by p_{sender} (Algorithm 1). If p_{sender} is correct it executes Algorithm 1 to broadcast a message m, then brb-delivers its own message at line 3. ◀

▶ **Lemma 13.** *Algorithm 2 verifies the* BRB-No-duplicity *Property.*

Proof.

- If p_{sender} is correct, p_{sender} brb-broadcasts one single message m (Algorithm 1), and by BRB-Validity (Lemma 10), all correct processes that do brb-deliver a message only brb-deliver m.
- If p_{sender} is Byzantine, consider two correct processes p_i and p_j (both necessarily different from p_{sender}) that each brb-deliver some message: p_i brb-delivers m_i and p_j brb-delivers m_j. We distinguish three cases depending on the lines at which p_i and p_j execute brb_deliver.
  - Case 1: Assume p_i and p_j both deliver their respective message at line 17 of Algorithm 2. Due to the condition at line 16, there exist two rounds R_i and R_j such that the following holds

    $$known_msgs_{i,R_i} = \{m_i\} \wedge \mathsf{certificate}_i(m_i, t+3-R_i), \text{ and}$$
    $$known_msgs_{j,R_j} = \{m_j\} \wedge \mathsf{certificate}_j(m_j, t+3-R_j).$$

    Without loss of generality, assume $R_i \leq R_j$. By Lemma 4, $\mathsf{certificate}_i(m_i, t+3-R_i) = \mathtt{true}$ implies that $m_i \in known_msgs_{j,R_i}$ at round R_i at p_j (since $R_i \leq R_j$ implies that p_j executes round R_i). $R_i \leq R_j$ further implies $known_msgs_{j,R_i} \subseteq known_msgs_{j,R_j}$ by definition of $known_msgs_{j,-}$, and the way $view_j$ is initialized and updated. $m_i \in known_msgs_{j,R_i}$ therefore implies that $m_i \in known_msgs_{j,R_j}$. Combined with $known_msgs_{j,R_j} = \{m_j\}$, this last statement yields $m_i = m_j$, proving the lemma.
  - Case 2: Assume p_i and p_j both brb-deliver their respective message at line 15 of Algorithm 2, during round $t+1$. Let us consider the two following sets, defined at round $t+1$:

    $$W_i = \big\{w \in \mathbb{N} \,\big|\, \exists m \in known_msgs_{i,t+1} : \mathsf{certificate}_i(m, w)\big\},$$

    and

    $$W_j = \big\{w \in \mathbb{N} \,\big|\, \exists m \in known_msgs_{j,t+1} : \mathsf{certificate}_j(m, w)\big\}.$$

    Consider $w \in W_i$. By Lemma 9, $\mathsf{certificate}_i(m, w) = \mathtt{true}$ implies $\mathsf{certificate}_j(m, w) = \mathtt{true}$ for p_j. Because of line 3 of the definition of certificate (Alg. 3), $\mathsf{certificate}_j(m, w) = \mathtt{true}$ implies that $m \in known_msgs_{j,t+1}$, and therefore that $w \in W_j$. Inverting p_i and p_j leads to $W_i = W_j$, and therefore to $wmax_i = wmax_j$.
    Using $wmax_i = wmax_j$, and following an identical reasoning on $candidate_msgs_i$ and $candidate_msgs_j$ produces $candidate_msgs_i = candidate_msgs_j$, and therefore that p_i and p_j brb-deliver the same message at line 15.
  - Case 3: Assume p_i brb-delivers m_i at line 17 of Algorithm 2, and p_j brb-delivers m_j at line 15 of the same algorithm. Due to the condition at line 16, there exists a round R_i such that the following holds

    $$known_msgs_{i,R_i} = \{m_i\} \wedge \tag{14}$$
    $$\mathsf{certificate}_i(m_i, t+3-R_i). \tag{15}$$

As in Case 2, let us consider the following set defined at round $t+1$ at p_j:

$$W_j = \{w \in \mathbb{N} \mid \exists m \in \mathit{known_msgs}_{j,t+1} : \mathsf{certificate}_j(m, w)\}.$$

Because of Lemma 9, $\mathsf{certificate}_i(m_i, t+3-R_i) = \mathtt{true}$ implies that $\mathsf{certificate}_j(m_i, t+3-R_i) = \mathtt{true}$ at p_j at round $t+1$. The condition at line 3 of the code of $\mathsf{certificate}$ (Algorithm 3) further means that $\mathsf{certificate}_j(m_i, t+3-R_i) = \mathtt{true}$ implies $m_i \in \mathit{known_msgs}_{j,t+1}$, and therefore that $t+3-R_i \in W_j$. This last inclusion yields that $\mathit{wmax}_j \geq t+3-R_i$ at line 13 of Algorithm 2.
m_j is brb-delivered by p_j at line 15. Therefore by construction $\mathsf{certificate}_j(m_j, \mathit{wmax}_j) = \mathtt{true}$. Due the condition at line 3 in the code of $\mathsf{certificate}$ (Alg. 3), $\mathit{wmax}_j \geq t+3-R_i$ and $\mathsf{certificate}_j(m_j, \mathit{wmax}_j) = \mathtt{true}$ yield $\mathsf{certificate}_j(m_j, t+3-R_i) = \mathtt{true}$. Using Lemma 4 this last statement implies that $m_j \in \mathit{known_msgs}_{i,R_i}$ at round R_i at p_i. Combined with (14), this leads to $m_j = m_i$, proving the case and concluding the lemma. ◀

▶ **Lemma 14.** *If p_{sender} is correct, correct processes brb-deliver the message m brb-broadcast by p_{sender} in at most $\mathsf{max}(2, t+3-c)$ rounds, where $c = n - f$ is the number of effective correct processes.*

Proof. If p_{sender} is correct, it broadcasts $m : p_{\text{sender}}$ to all correct processes (line 2 or Alg. 1), and brb-delivers its own message in round 1. Every correct process p_j other than p_{sender} receives $m : p_{\text{sender}}$ in round 1, and broadcasts $m : p_{\text{sender}} : p_j$ in round 2. At the end of round 2, a correct process p_i has therefore received at least $c-1$ length-2 signature chains for m:

$$\{m : p_{\text{sender}} : p_j \mid p_j \in \Pi_c \setminus \{p_{\text{sender}}\}\} \subseteq \mathit{view}_i[2], \tag{16}$$

where Π_c is the set of correct processes. In round 2, we therefore have

$$\forall q \in \Pi_c \setminus \{p_{\text{sender}}, p_i\},\ m : p_{\text{sender}} : q \in \mathsf{truncate}_2\left(\mathit{view}_i[2]\right).$$

As at round 2, $\mathit{view}_i[r'] = \emptyset$ for all $r' > 2$, this leads to

$$\forall q \in \Pi_c \setminus \{p_{\text{sender}}, p_i\},\ m : p_{\text{sender}} : q \in \mathsf{truncate}_2\left(\textstyle\bigcup_{r' \in [2..t+1]} \mathit{view}_i[r']\right)$$

As $\mathsf{truncate}_{t+2-c}(p_i)$ is either the empty sequence or p_i, this further leads to

$$\Pi_c \setminus \{p_{\text{sender}}, p_i\} \subseteq \left\{q \in \Pi \setminus \mathsf{truncate}_{t+2-c}(p_i) \,\middle|\, m : p_{\text{sender}} : q \in \mathsf{truncate}_2\left(\textstyle\bigcup_{r' \in [2..t+1]} \mathit{view}_i[r']\right)\right\}.$$

Because p_{sender} and p_i are correct, $|\Pi_c \setminus \{p_{\text{sender}}, p_i\}| = c-2$ and the previous inclusion implies

$$\left|\left\{q \in \Pi \setminus \mathsf{truncate}_{t+2-c}(p_i) \,\middle|\, m : p_{\text{sender}} : q \in \mathsf{truncate}_2\left(\textstyle\bigcup_{r' \in [2..t+1]} \mathit{view}_i[r']\right)\right\}\right| \geq c-2. \tag{17}$$

Because p_i is correct, (16) further yields

$$m : p_{\text{sender}} : p_i \in \mathit{view}_i[2]. \tag{18}$$

By choosing $r_i = 2$ and $\gamma_i = p_i$, (17) and (18) render true line 3 of $\mathsf{certificate}_i(m, c)$ (Alg. 3) at round 2. In other words, p_i (and thus every correct process other than p_{sender}) observes a certificate of weight c for m at round 2. By definition of the $\mathsf{certificate}$ function, and

construction of $view_i$, $\mathsf{certificate}_i(m,c) = \mathtt{true}$ at round 2 implies that $\mathsf{certificate}_i(m,c)$ remains true during the rest of p_i's execution. In addition, as p_{sender} is correct and signatures are secure, $known_msgs_{i,R}$ does not contain any other message than m. Therefore, if p_i does not brb-deliver m earlier, at the latest at round $R_c = \mathsf{max}(2, t+3-c)$ the condition of line 16 becomes true, and p_i brb-delivers m during the same round. ◀

▶ **Lemma 15.** *Algorithm 2 verifies the* BRB-Global-delivery *property.*

Proof.

- If p_{sender} is correct, then using Lemma 14 all correct processes execute brb_deliver, the lemma is verified.
- If p_{sender} is Byzantine, consider p_i and p_j two correct processes ($\{p_i, p_j\} \cap \{p_{\text{sender}}\} = \emptyset$) and assume p_i brb-delivers some message m. Whether p_i brb-delivers m at line 15 or at line 17, the brb-delivery implies that p_i observes $\mathsf{certificate}_i(m,w) = \mathtt{true}$ for some $w \in \mathbb{N}$ at some round $R_i \leq t+1$. Let r_i and γ_i denote a round and a process sequence that render true the condition at line 3 for p_i in the definition of $\mathsf{certificate}_i$ (Algorithm 3) at round R_i. Since at round R_i, $\forall r' \in [R_i+1..t+1] : view_i[r'] = \emptyset$. Line 3 further implies that

$$\begin{aligned} \mathsf{truncate}_2(m : p_{\text{sender}} : \gamma_i) &\in \mathsf{truncate}_2(view_i[r_i]) \\ &\subseteq T_{2,i}[r_i] \qquad \text{since } r_i \geq 2 \text{, and by definition of } T_{2,i}[r_i]. \end{aligned}$$

  Using Corollary 8 we have $T_{2,i}[r_i] \subseteq T_{2,j}[t+1]$ and therefore $\mathsf{truncate}_2(m : p_{\text{sender}} : \gamma_i) \in T_{2,j}[t+1]$.
  Using the definition of $T_{2,j}[t+1]$ (Equation 1), $\mathsf{truncate}_2(m : p_{\text{sender}} : \gamma_i) \in T_{2,j}[t+1]$ implies that

$$\exists r_j \in [2..t+1], \exists \pi_j \in view_j[r_j] : \mathsf{truncate}_2(\pi_j) = \mathsf{truncate}_2(m : p_{\text{sender}} : \gamma_i)$$

  Because p_j is correct, it only accepts valid signature chains, which implies that π_j is of the form $\pi_j = m : p_{\text{sender}} : \gamma_j$ for some $\gamma_j \in \Pi^{r_j - 1}$ (since $\pi_j \in view_j[r_j]$). $m : p_{\text{sender}} : \gamma_j \in view_j[r_j]$ implies $\mathsf{certificate}_j(m,2) = \mathtt{true}$ at p_j at round r_j (since line 3 of Algorithm 3 is trivially true for $w = 2$), and $m \in known_msgs_{j,r_j}$.
  If we assume p_j does not brb-deliver any message before round $t+1$, the fact that $\mathsf{certificate}_j(m,2) = \mathtt{true}$ at p_j at round $r_j \leq t+1$ means that $\mathsf{certificate}_j(m,2) = \mathtt{true}$ at p_j at round $t+1$, due to the definition of the function certificate (Algorithm 3). Similarly, since $known_msgs_{j,r_j} \subseteq known_msgs_{j,t+1}$, $m \in known_msgs_{j,r_j}$ implies $m \in known_msgs_{j,t+1}$. The two facts $m \in known_msgs_{j,t+1}$ and $\mathsf{certificate}_j(m,2) = \mathtt{true}$ at p_j at round $t+1$ imply $W_j \neq \emptyset$ at line 12 of Algorithm 2, and therefore that $candidate_msgs_j \neq \emptyset$, leading p_j to brb-deliver some message at line 15. ◀

A.3 Numerical comparison

Assuming $t < 3/4 \times n$, and that only $\lfloor t/2 \rfloor$ processes have effectively been compromised (equivalently, $c = n - \lfloor t/2 \rfloor$), the good-case latency of our algorithm outperforms the optimized version of [27] presented in [4] up to $n \leq 43$, and is at least as good up to $n \leq 51$. (This claim follows from an exhaustive computation of the values of $\mathsf{max}(2, t+3-c)$ and $\lceil \frac{n}{n-t} \rceil + \lfloor \frac{n}{n-t} \rfloor$ over $n \in [3..52], t \in [1..\lfloor 3/4 \times n \rfloor - 1]$, with $c = n - \lfloor t/2 \rfloor$.)

It follows that the proposed algorithm is particularly well suited to small- and medium-size synchronous distributed systems.

Polynomial-Time Verification and Testing of Implementations of the Snapshot Data Structure

Gal Amram
Ben Gurion University of the Negev, Beer-Sheva, Israel
IBM Research, Haifa, Israel

Avi Hayoun
Ben-Gurion University of the Negev, Beer-Sheva, Israel

Lior Mizrahi
Ben-Gurion University of the Negev, Beer-Sheva, Israel

Gera Weiss
Ben-Gurion University of the Negev, Beer-Sheva, Israel

Abstract

We analyze correctness of implementations of the snapshot data structure in terms of linearizability. We show that such implementations can be verified in polynomial time. Additionally, we identify a set of representative executions for testing and show that the correctness of each of these executions can be validated in linear time. These results present a significant speedup considering that verifying linearizability of implementations of concurrent data structures, in general, is EXPSPACE-complete in the number of program-states, and testing linearizability is NP-complete in the length of the tested execution. The crux of our approach is identifying a class of executions, which we call *simple*, such that a snapshot implementation is linearizable if and only if all of its simple executions are linearizable. We then divide all possible non-linearizable simple executions into three categories and construct a small automaton that recognizes each category. We describe two implementations (one for verification and one for testing) of an automata-based approach that we develop based on this result and an evaluation that demonstrates significant improvements over existing tools. For verification, we show that restricting a state-of-the-art tool to analyzing only simple executions saves resources and allows the analysis of more complex cases. Specifically, restricting attention to simple executions finds bugs in 27 instances, whereas, without this restriction, we were only able to find 14 of the 30 bugs in the instances we examined. We also show that our technique accelerates testing performance significantly. Specifically, our implementation solves the complete set of 900 problems we generated, whereas the state-of-the-art linearizability testing tool solves only 554 problems.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Theory of computation → Concurrent algorithms

Keywords and phrases Snapshot, Linearizability, Verification, Formal Methods

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.5

Related Version *Full Version*: `https://github.com/hayounav/Thesis_experiments/blob/main/snapshot verification and testing/Polynomial_time_verification_of_snapshot_implementations_DISC_.pdf`

Supplementary Material *Software (Source Code)*: `https://github.com/hayounav/Thesis_experiments`; archived at `swh:1:dir:10be8bad714e0da40fec5d0a1a6aa34c550ccc33`

Funding This research was partially funded by grant no. 2714/19 from the Israel Science Foundation and by the Lynn and William Frankel Center for Computer Science at Ben-Gurion University.

1 Introduction

As concurrency is very effective for accelerating the performance of computer programs, there is much scientific research and practical attention on the design, implementation, and verification of data structures that allow parallel access. We focus on the well-known *snapshot*

© Gal Amram, Avi Hayoun, Lior Mizrahi, and Gera Weiss;
licensed under Creative Commons License CC-BY 4.0

data structure which is an essential building block of distributed arrays [4, 7, 9]. This data structure allows asynchronous processes to write values to a shared array of single-writer registers, by executing update operations, and to take instantaneous snapshots of the array values, by executing scan operations. It is useful for allowing processes to share information while maintaining a correct joint view of the data.

For proving the correctness of implementations of the snapshot data structure, we consider the standard *linearizability* [31] condition. Roughly speaking, linearizability is the requirement that for every execution of a given implementation, the procedure executions can be ordered linearly such that (a) the resulting linear order is consistent with the definition of the data structure; (b) it preserves the precedence of procedure executions in time. In the specific case of snapshot, the definition of the data structure is that a sequential (linear) execution is correct if every scan operation reports the value written by the last update operation of each of the processes. Linearizability is widely accepted as a correctness criterion since, effectively, it formulates the requirement that procedure executions are seen to a user as if they were executed one after the other (i.e., atomically).

Automatic verification of linearizability is known to be computationally expensive. The verification of finite-state implementations is EXPSPACE-complete in the number of program states [5, 29] and undecidable for infinite-state implementations [14]. Testing linearizability, i.e., deciding whether a given execution is linearizable, is NP-complete [28]. These complexities do not stop the community from pursuing effective verification and testing techniques, because it is very difficult to provide correct implementations of concurrent data structures; bugs have been found in both academic and deployed implementations [18, 20, 42]. These made it clear that there is an acute necessity for reliable verification and testing techniques.

One commonly used technique is the *linearization points* based verification approach, which often does not work in the case of snapshot. A linearization point of a procedure execution is an action that represents the moment at which the procedure "actually occurs". Once fixed linearization points are identified, verifying linearizability becomes PSPACE-complete [14]. Unfortunately, snapshot implementations do not usually admit fixed linearization points (e.g., all twelve published implementations listed in [35] do not admit such points). Researchers also suggested using linearization points as an optimization: ask the user to provide them (whether fixed or conditional) and use this information to accelerate verification [3, 6, 13, 50]. However, practice shows that it is difficult to find and specify the linearization points of snapshot implementations, even in a conditional manner. One difficulty is that the linearization points of scan operations often belong to other, parallel, procedure executions (see [4, 11, 46]).

In this paper, we propose an effective polynomial-time technique for verifying snapshot implementations, and an effective linear-time technique for testing snapshot executions. The crux of our techniques is an optimization approach that exponentially reduces the number of reachable program states. Specifically, we prove that if an algorithm is data-independent [54] then, in order to verify its correctness, it suffices to consider only a small fraction of its executions which we call *simple*.

The simple executions that we focus on are those in which:

1. All but two processes invoke only update(v_0) and scan operations, where v_0 is the initial value of the array segments. In other words, $n-2$ of the n processes are not allowed to change the initial value in their segments;
2. Each of the two remaining processes may only change their data value once, to a predetermined value: it executes only update(v_0) and scan operations up to an arbitrary point in the execution, after which it transitions to executing only update(v_1) and scan operations, where $v_1 \neq v_0$ are fixed data values.

The focus on simple executions reduces the number of reachable states significantly, as $n-2$ entries of the array are essentially constants (see Section 3).

After showing that it is enough to verify the correctness of simple executions, we continue and show that every non-linearizable simple execution falls into one of three categories that we identify. Moreover, we show that each of these three possible bug patterns can be recognized by an automaton with at most n states and n^2 transitions (see Section 4). This enables verifying linearizability of snapshot implementations via a reachability check applied to the graph product of the implementation and the automatons where the target states of the reachability are the automatons' accepting states. As there are $O(n^2)$ combinations to choose the two excluded processes, snapshot implementations can be verified with this method in time $O(mn^4)$ where m is the number of reachable states via simple executions (which is significantly smaller than the number of reachable states via all executions). Furthermore, by feeding a simple execution to these automatons, an execution of length l can be tested in time $O(l)$. As it is sufficient to consider simple executions, this effectively means that snapshot executions can be tested in linear-time (see Section 5).

We have implemented and evaluated the proposed verification and testing techniques and ran them against state-of-the-art tools. For verification, we compared with the PAT [44] model checker. The results show that our approach allows deeper exploration of implementations from the literature. This allowed us to detect 27 of 30 inserted bugs, compared to 16 found by the baseline method. Furthermore, we managed to verify an algorithm by Bowman [17] for three and four processes, whereas the baseline method failed to do so. For testing, we compared with the linearizability testing tool proposed by Lowe [41]. The results show that our testing technique is robust and scalable and that it can cope with much longer histories than the baseline (see Section 7).

Due to lack of space, we give proof sketches and skip technical details. We provide a full version with complete proofs, and means to reproduce the experiments in the paper supporting materials [49].

2 Preliminaries

This section presents definitions and notations used throughout this paper. We provide further definitions, required for all complete proofs, and extended discussions in the supporting materials [49, Appendix A].

Let *Vals* be an infinite set of abstract data values, and let $v_0 \in \mathit{Vals}$ be the distinguished value used to initialize the segments of a snapshot. For $n \in \mathbb{N}$, let $p_0, \ldots, p_{n-1}$ be processes. We model an execution of a snapshot algorithm by the processes as a sequence of actions. Among the actions the processes perform, we are interested in the invocations and responses of procedure executions. For process p_i and data values $u, u_0, \ldots, u_{n-1}$, $\mathit{inv}.\mathsf{update}_i(u)$, $\mathit{res}.\mathsf{update}_i$, $\mathit{inv}.\mathsf{scan}_i$, $\mathit{res}.\mathsf{scan}_i(u_0, \ldots, u_{n-1})$ are p_i*-actions*. Let Σ be the set of all such actions.

Throughout the paper, we refer to these actions using general terms such as: an update invocation, a scan response, a p_i-invocation etc., which are defined in a straightforward manner. For example, we may say that the action $\mathit{res}.\mathsf{scan}_i(u_0, \ldots, u_{n-1})$ is a scan response, or a p_i-action, etc.

A *history* is a word h over Σ that exhibits the following properties:

1. For every process p_i, the first p_i-action in h, if any, is a p_i-invocation.
2. For every p_i-update (respectively, scan) invocation in h, the following p_i-action in h, if any, is a p_i-update (respectively, scan) response.
3. For every p_i-response in h, the following p_i-action, if any, is a p_i-invocation.

$U_1 = \mathsf{update}_0(x)$
$U_2 = \mathsf{update}_1(y)$ $S_1 = \mathsf{scan}_1(x, y)$ $S_2 = \mathsf{scan}_1$
time
$inv.\mathsf{update}_0(x)$ $inv.\mathsf{update}_1(y)$ $res.\mathsf{update}_1$ $inv.\mathsf{scan}_1$ $res.\mathsf{scan}_1(x, y)$ $inv.\mathsf{scan}_1$

Figure 1 A linearizable history. U_2, S_1 are complete, while U_1, S_2 are pending ops.

An operation is an execution of an update/scan procedure. We identify operations in histories with their invocation and response actions. Operations that do not return are identified by their invocation alone.

A *complete operation* in a history $h = \alpha_0 \cdots \alpha_m$ is a pair of actions, (α_k, α_l), where $k < l$, α_k is an update_i (respectively, scan_i) invocation, α_l is an update_i (respectively, scan_i) response, and there is no p_i-action in between. A *pending operation* in h is a single action, (α_k), where α_k is a p_i-invocation, and there is no p_i-action that follows α_k in h.

Similarly to actions, we refer to operations using general terms: an operation O is, e.g., a p_i-operation, an update operation, a $\mathsf{scan}_i(u_0, \ldots, u_{n-1})$ operation, etc. In a history h, for an $\mathsf{update}(u)$ operation U, we write $val_h(U) = u$, and for a $\mathsf{scan}(u_0, \ldots, u_{n-1})$ operation S and $i < n$, we write $val_{h:i}(S) = u_i$ and $val_h(S) = (u_0, \ldots, u_{n-1})$.

For a complete operation $A = (\alpha_k, \alpha_l)$ and an operation $B \in \{(\alpha_m, \alpha_t), (\alpha_m)\}$ in a history $h = \alpha_0\alpha_1 \cdots$, we write $A <_h B$ if $l < m$. Clearly, $<_h$ is a partial order over the operations in h, in which pending operations are maximal elements. Figure 1 illustrates a two-process history with pending and complete operations.

Linearizability [31] is the standard correctness condition for concurrent data structures. Roughly speaking, a history h is linearizable if the partial ordering $<_h$ can be extended to a linear ordering that satisfies the sequential specification of the snapshot data structure. That is, each scan operation S returns in each entry i the value written by the maximal update_i operation that precedes it. The extension should include all complete operations, and each pending operation is either completed or omitted.

We now turn to define the linearizability condition formally:

▶ **Definition 1.** *A history h is linearizable if it can be extended into a history h' by appending zero or more response events to h, such that there exists a linear ordering $\prec_{h'}$ of the complete operations in h' that satisfies the following conditions:*

1. *For $A, B \in h$, if $A <_h B$, then $A \prec_{h'} B$.*
2. *If $S \in h'$ is a* scan *operation and $U_i \in h'$ is the maximal* update_i *operation such that $U_i \prec_{h'} S$, then $val_h(U_i) = val_{h:i}(S)$. If no* update_i *operation precedes S in h', then $val_{h:i}(S) = v_0$.*

Any $\prec_{h'}$ that satisfies these conditions is said to be a linearization *of h.*

▶ **Example 2.** The history depicted in Figure 1 is linearizable by the order $U_1 \prec_h U_2 \prec_h S_1$. To obtain a linearization, we completed the pending operation U_1, as its value is read by S_1. However, we chose to omit the pending scan operation S_2.

Our main goal is to analyze the linearizability of snapshot algorithms, defined as follows:

▶ **Definition 3** (Snapshot Linearizability)**.** *A snapshot algorithm is linearizable if all of its histories are linearizable.*

The data independence property, proposed by Wolper [54], roughly means that the behavior of an algorithm does not depend on the data values passed as arguments to the procedure executions. The formal definition employs the notion of a *renaming*: a function $f\colon Vals \longrightarrow Vals$. An algorithm is data-independent if for a every history h of the algorithm and a renaming f: (1) the history $f(h)$, obtained by replacing each data value u with $f(u)$, is also a history of the algorithm; and (2) if $h = f(h')$, then h' is a history of the algorithm. See full version [49] for more details.

Data independence is natural to assume, as snapshot implementations synchronize accesses to a shared resource and thus are expected to be value-agnostic. This is substantiated by all twelve different published implementations [4, 7–11, 26, 34–36, 46] listed in [35].

Finally, a history is *differentiated* if no two update operations in it were invoked with the same data value.[1] Abdulla et al. [1] showed that it is sufficient to consider *differentiated* histories to prove linearizability of data-independent algorithms.

3 Simple Histories

In this section, we identify a set of histories, which we name *simple*. We then prove that a data-independent snapshot implementation is linearizable if and only if all of its simple histories are linearizable. Therefore, this section shows that it is sufficient to consider only some histories to determine the linearizability of data-independent snapshot implementations.

In a simple history, the update operations are invoked with only two distinct values. The first is the initial value v_0, and without loss of generality, we take some other $v_1 \in Vals$ as the second value. All but two processes invoke only update(v_0) and scan operations. The remaining two execute only update(v_0) and scan operations, and at some (possibly different) point, each of the two processes shifts to executing only update(v_1) and scan operations.

▶ **Definition 4** (Simple histories). *A history h of n processes is (i, j)-simple for $i < j < n$, if there are $r_i, r_j \in \mathbb{N}$ such that the following conditions hold:*

1. *Let U be the r-th* update_i *operation in h. If $r < r_i$, then U is an* $\mathsf{update}_i(v_0)$ *operation, and if $r \geq r_i$, then U is an* $\mathsf{update}_i(v_1)$ *operation.*
2. *In the same way, let U be the r-th* update_j *operation in h. If $r < r_j$, then U is an* $\mathsf{update}_j(v_0)$ *operation, and if $r \geq r_j$, then U is an* $\mathsf{update}_j(v_1)$ *operation.*
3. *Any* update_k *operation is an* $\mathsf{update}_k(v_0)$ *operation, if $k \notin \{i, j\}$.*

A history h is simple if it is (i, j)-simple for some $i < j < n$.

We are ready to prove the sufficiency of focusing on simple histories. We provide a proof sketch below, and give a rigorous proof in the full version [49].

▶ **Theorem 5.** *A data-independent snapshot algorithm $\mathscr{S}nap$ is linearizable if and only if all its simple histories are linearizable.*

Proof sketch. Anderson's shrinking lemma identifies five properties that are equivalent to the linearizability of a snapshot history [8]. To prove the non-trivial direction of theorem 5 ('if'), we assume that $\mathscr{S}nap$ is not linearizable. Consider some non-linearizable differentiated history h. Since $\mathscr{S}nap$ is not linearizable, h violates (at least) one of the shrinking lemma's properties. Based on the violated property, we construct a renaming $f\colon Vals \to \{v_0, v_1\}$, and apply it to h to obtain a non-linearizable simple history. ◀

[1] For generating differentiated histories, a minor modification is required: we should allow different initial values for the array segments. See comment in full version [49, Appendix B]

Figure 2 The k-th interval of S, $I_k(S)$.

▶ Remark 6. In the context of simple histories, scan operations return v_0 in all entries except for entries i and j. Thus, for the remainder of this paper, we use $res.\mathsf{scan}_k(u_i, u_j)$ as shorthand for $res.\mathsf{scan}_k(v_0, \ldots, v_0, u_i, v_0, \ldots, v_0, u_j, v_0, \ldots, v_0)$.

▶ Remark 7. From this point on, for readability, we will use 0 and 1 instead of v_0 and v_1, respectively.

4 A Simple Condition for the Linearizability of Simple Histories

In this section, we formulate three properties that are equivalent to the linearizability of an (i, j)-simple history. We then show that the negation of each property is regular, and present a construction of a matching automaton. Before providing our properties (in upcoming Theorem 10), we discuss each intuitively and explain why it is mandatory for linearizability.

Property 1: No Inversion. Assume that a scan operation S_1 returns 0 at the ith entry (for example), while S_2 returns 1. This indicates that S_2 read a more recent value from the ith segment. Hence, in any linearization, S_1 must precede S_2. As the same reasoning goes for the jth entry, it is forbidden for S_1 to return 0 and 1 at the ith and jth entries, while S_2 returns the opposite values.

Property 2: Non-Decreasing. If a scan operation S_1 precedes a scan operation S_2, then S_2 must obtain more recent values from all array segments. Therefore, it is forbidden for S_1 to return 1 in entry $k \in \{i, j\}$, while S_2 returns 0 in its kth entry.

Property 3: Appropriateness. We require that for each scan operation there are "appropriate" update operations, U_i by p_i and U_j by p_j, that we can linearize before S. "Appropriate" means that the next three conditions hold.

First condition. The timings of the update operations must not prevent them from being linearized before S. For example, they must not succeed S. Formally, we require that they belong to the *interval of* S, defined below and illustrated in Figure 2:

▶ **Definition 8.** *Let S be a* scan *operation in a history h, and let $k < n$. The kth interval of S, denoted $I_k(S)$, is the set of* update_k *operations $U \in h$ such that:*

1. $\neg(S <_h U)$.
2. *There is no* update_k *operation U' such that $U <_h U' <_h S$.*

Second condition. The values of the update operations U_i and U_j are the values returned by S in its corresponding entries.

Third condition. There is no, e.g., update_i operation between U_i and U_j. This is because the existence of such an update_i operation, say $U_i < U_i' < U_j$, would prevent us from linearizing both U_i and U_j before S.

We formalize the notion of appropriateness in the following definition:

▶ **Definition 9.** *Let S be a complete* scan *operation in an (i, j)-simple history h. A pair (U_i, U_j) where U_i is an* update_i *operation and U_j is an* update_j *operation, is said to be S-appropriate, if:*
1. *$U_i \in I_i(S)$ and $U_j \in I_j(S)$.*
2. *$(val_h(U_i), val_h(U_j)) = (val_{h:i}(S), val_{h:j}(S))$.*
3. *There is no* update_i *operation U_i' such that $U_i < U_i' < U_j$, and there is no* update_j *operation U_j' such that $U_j < U_j' < U_i$.*

So far, we have presented our properties and explained intuitively why they form a necessary condition for linearizability: i.e., why their negation prevents linearizability. The main theorem of this section asserts a much stronger claim: these properties also constitute a sufficient condition for linearizability. We provide a proof for Theorem 10 in the full version [49, Appendix D].

▶ **Theorem 10.** *An (i, j)-simple history h is linearizable if and only if the following properties hold.*

No Inversion. *There are no complete* scan *operations S_1 and S_2 in h such that $(val_{h:i}(S_1), val_{h:j}(S_1)) = (0, 1)$ and $(val_{h:i}(S_2), val_{h:j}(S_2)) = (1, 0)$.*

Non-Decreasing. *If S_1 and S_2 are two complete* scan *operations in h such that $S_1 <_h S_2$, then $val_{h:i}(S_1) \le val_{h:i}(S_2)$ and $val_{h:j}(S_1) \le val_{h:j}(S_2)$.*

Appropriateness. *For each complete* scan *operation S in h, there exists an S-appropriate pair of update operations.*

4.1 Detecting Incorrect Simple Histories

Finally, we show that the properties of Theorem 10 can be detected by an NFA. We provide here proof sketches for most claims, and full proofs for all claims in the full version [49, Appendix E].

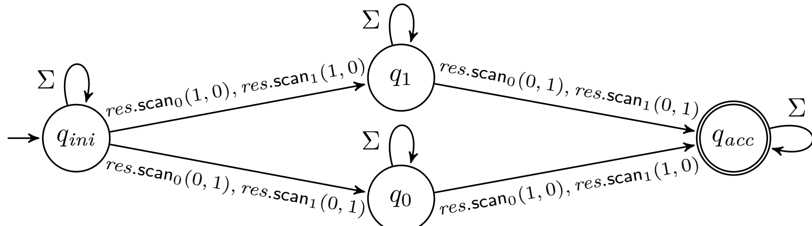
▶ **Theorem 11.** *For $i < j < n$, there exists an automaton M with $O(n)$ states and $O(n^2)$ transitions, such that an (i, j)-simple history h is not linearizable if and only if $h \in L(M)$.*

To prove Theorem 11, we construct automatons that detect violations of the three properties presented in Theorem 10.

4.1.1 Detecting Violations of No-Inversion

▶ **Proposition 12.** *There exists an automaton M_1 such that, for any (i, j)-simple history h, h violates No-Inversion if and only if $h \in L(M_1)$. Moreover, M_1 has $O(1)$ states and $O(n)$ transitions.*

Proof sketch. We demonstrate the construction for the case that $n = 2$:

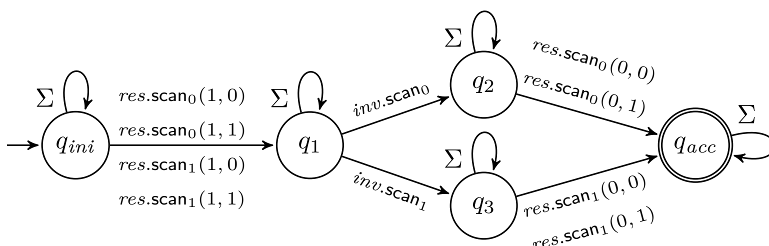


◀

4.1.2 Detecting Violations of Non-Decreasing

▶ **Proposition 13.** *There exists an automaton M_2 such that, for any (i,j)-simple history h, h violates Non-Decreasing if and only if $h \in L(M_2)$. Moreover, M_2 has $O(n)$ states and $O(n^2)$ transitions.*

Proof sketch. As an illustrative demonstration, we present below a simpler automaton. It detects the existence of a violation of Non-Decreasing, for $n = 2$, and $S_1 < S_2$ where $val_{h:i}(S_1) = 1$.



◀

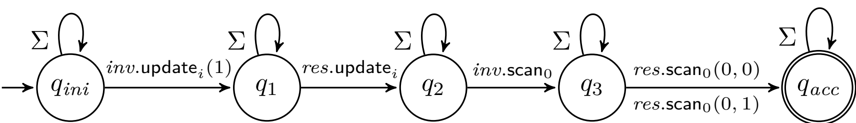
4.1.3 Detecting Violations of Appropriateness

It remains to construct an automaton that accepts all (i,j)-simple histories that violate Appropriateness. To this end, we reformulate Appropriateness as a regular safety property; a word is rejected if and only if it has a “bad”-prefix.

▶ **Lemma 14.** *Let h be an (i,j)-simple history, and let S be a complete* scan *operation in h. For $l \in \{i,j\}$, let F_l be the first* $\mathsf{update}_l(1)$ *operation in h, if exists. Then, there is no S-appropriate pair in h if and only if any of the following holds:*
1. *For $l \in \{i,j\}$, F_l exists, $val_{h:l}(S) = 0$, and $F_l <_h S$.*
2. *For $l \in \{i,j\}$, $val_{h:l}(S) = 1$, and either $S <_h F_l$ or F_l doesn't exist.*
3. *$(val_{h:i}(S), val_{h:j}(S)) = (0,1)$ and $F_i <_h F_j$.*
4. *$(val_{h:i}(S), val_{h:j}(S)) = (1,0)$ and $F_j <_h F_i$.*

▶ **Proposition 15.** *There exists an automaton M_3 such that, for any (i,j)-simple history h, h violates Appropriateness if and only if $h \in L(M_3)$. Moreover, M_3 has $O(n)$ states and $O(n^2)$ transitions.*

Proof sketch. The automaton is a “union” of four automatons, such that the kth automaton checks whether there exists a complete scan operation S for which the kth case of Lemma 14 holds. Below, we provide an automaton that identifies the first case where $l = i$ and S is a scan_0 operation (for $n = 2$). Hence, in fact, the first case is a union of $2n$ automatons. We leave it for the reader to verify that (rather simple) automatons exist for all other cases.



◀

▶ **Corollary 16.** *Theorem 11 is trivially correct by propositions 12, 13, and 15.*

5 Verifying and Testing Linearizability

The results of the previous section allow us to both verify data-independent snapshot implementations, and test the linearizability of simple snapshot histories, in polynomial time.

For verifying a data-independent snapshot implementation, by Theorem 5, it is sufficient to verify that all of its simple histories are linearizable. Theorem 11 allows one to apply model checking of regular properties [12, Chapter 4.2], i.e., one can check whether the implementation admits an (i, j)-simple history accepted by the automaton (and thus not linearizable). As there are $O(n^2)$ valuations for i and j, our first key result follows:

▶ **Theorem 17** (Polynomial-time verification). *Let $\mathscr{S}nap$ be a data-independent snapshot algorithm such that only finitely many states of $\mathscr{S}nap$ are reachable by its simple histories. Determining the linearizability of $\mathscr{S}nap$ is decidable in $O(mn^4)$ time, where m is the size of an automaton that accepts all simple histories of $\mathscr{S}nap$.*

Moreover, Theorem 11 enables the testing of simple histories efficiently, by feeding the automaton of Theorem 11 (or its determinization) with the (i, j)-simple histories to be tested. The automaton will report acceptance once it identifies a non-linearizable prefix of the input history h (which testifies that h is not linearizable). Hence, our second key result follows.

▶ **Theorem 18** (Linear-time testing). *For $i < j < n$, (i, j)-simple histories can be tested in linear-time, i.e., in time $O(|h|)$.*

6 Optimization: Omitting Redundant Commands

The focus on simple histories yields an optimization that can significantly reduce the state space of an examined snapshot algorithm, speeding up its verification. The optimization relies on the observation that in the executions of (i, j)-simple histories, some commands are vacuous. To elaborate, assume that register R stores the data value of process p_k, $k \neq i, j$. In all executions of (i, j)-simple histories, R will only ever store the value 0. Thus, read and write commands from and to R can be ignored, reducing the possible values of the program counters. In some cases, we can even ignore R altogether, reducing the number of registers.

We use Bowman's obstruction-free snapshot algorithm [17] (Algorithm 1) to demonstrate the optimization. During an update operation, process p_k writes its new value to register $A[k]$ (line 3). During a scan operation, the values stored in $A[0], \ldots, A[n-1]$ are read into the local variables $a[0], \ldots, a[n-1]$ (lines 7-8). Let $i < j$ be two process ids, and consider executions of (i, j)-simple histories of Algorithm 1. In such executions, every write command to register $A[k]$, $k \notin \{i, j\}$, writes 0. Hence, we may ignore and omit all registers $A[k]$, $k \notin \{i, j\}$. This yields a simplified version of the algorithm, as shown in Algorithm 2, which has a substantially smaller state space than Algorithm 1, as it employs fewer registers. Since we omitted only vacuous commands (i.e. commands that always write and read 0) Algorithm 2 is linearizable if and only if all of Algorithm 1's (i, j)-simple histories are linearizable.

7 Implementation and Evaluation

In this section, we describe implementations of the procedures described in Section 5, and the experiments we performed to evaluate their efficiency. We provide the means to reproduce all experiments in the paper's supporting materials [49].

Algorithm 1 Unoptimized algorithm.

```
1:  procedure update_k(v)
2:      Active ← ⊥
3:      A[k] ← v

4:  procedure scan_k
5:      repeat
6:          Active ← k
7:          for ℓ = 0, . . . , n−1 do
8:              a[l] ← A[l]
9:      until Active = k
10:     return (a[0], . . . , a[n−1])
```

Algorithm 2 Optimized for (i, j)-simple executions.

```
1:  procedure update_k                              ▷ k ∉ {i, j}
2:      Active ← ⊥

3:  procedure update_r(v)                           ▷ r ∈ {i, j}
4:      Active ← ⊥
5:      A[r] ← v

6:  procedure scan_q                                ▷ q < n
7:      repeat
8:          Active ← q
9:          a[i] ← A[i]
10:         a[j] ← A[j]
11:     until Active = q
12:     return (0, . . . , 0, a[i], 0, . . . , 0, a[j], 0, . . . , 0)
```

Figure 3 Illustration of the redundant command omission optimization with Bowman's algorithm.

7.1 Implementation of our Verification Procedures

We used the model checker PAT [44] as the basis for our two verification approaches. PAT contains a system for checking the linearizability of a given concurrent algorithm against an abstract specification, via refinement [38, 40]. We made use of this system in our first verification approach: we encoded known snapshot algorithms from the literature (listed in subsection 7.4), modified to admit only simple histories. We provided a matching abstract simple-history snapshot specification.

For our second approach, we encoded the automaton from Theorem 11 in PAT. As that automaton is a union of several automatons, we treated each one as a separate process, and encoded the union as the parallel composition of these processes. We exploited PAT's reachability checker to encode the accepting states of the automaton. We then asked PAT to check whether the algorithms listed in subsection 7.4 admit any simple histories that are accepted by the automaton.

We note three sources of possible errors in our implementations: (1) We could have encoded the snapshot algorithms incorrectly. (2) We could have encoded the automatons or the abstract specification incorrectly. (3) PAT itself may have bugs. To mitigate the first two threats, we used PAT's linearizability system to ensure that the algorithms we encoded are linearizable, that we manage to find several artificially-inserted bugs, and that the reachability approach agrees with PAT's standard refinement approach. We did not take steps to mitigate the third threat, but as PAT is a widely used model checker which has itself been partially model-checked [48], our confidence in its correctness is high., our confidence in its correctness is high.

7.2 Implementation of our Testing Procedure

The testing procedure we implemented receives an (i, j)-simple history, and runs it through an implementation of the automaton described in Theorem 11. The tool announces whether the automaton accepts the history, indicating it is not linearizable, or it rejects the history, indicating it is linearizable.

To validate that our implementation has no bugs, we generated hundreds of random simple histories, both linearizable and non-linearizable, and ensured our implementation classified them correctly.

7.3 Research Questions

We start with research questions related to our verification technique. As we propose a model checking approach, although polynomial, it still suffers from the state explosion problem [23]. This holds since the algorithms we check admit an enormous number of states, even when we restrict ourselves to simple histories. Model-checking approaches are mainly evaluated based on their feasibility; their ability to verify correctness/find bugs, perhaps only up to a reasonable depth, measured in the number of operations each process executes, with real-world resources: realistic time and space and limitations. Hence, we formulate the following research questions:

RQ1 Does the focus on simple histories help to prove/disprove correctness, in terms of feasibility/depth to be processed?

RQ2 Is our polynomial-time technique efficient for proving/disproving correctness, in terms of feasibility/depth to be processed?

We use the following research question to evaluate our testing technique:

RQ3 Is our testing technique efficient, in terms of feasibility, and time and space consumption?

7.4 Corpus

To address RQ1 and RQ2, we constructed a corpus for our experiments that includes several snapshot algorithms from the literature: An obstruction-free [30] algorithm by Bowman [17], denoted BOWMAN; A snapshot algorithm by Jayanti [35], denoted JAYANTI; The bounded and unbounded versions of Afek et al. [4], denoted AFEK1 and AFEK2, respectively; and A snapshot algorithm by Riany et al [46], denoted RIANY.

For each algorithm and $n \in \{3, 4, 5, 6\}$ processes, we encoded the original version (denoted "full"), as well as a modified version which generates only $(0, 1)$-simple histories, with the optimization detailed in Section 6 (denoted "simple-only").Then, for $n \in \{3, 4, 5, 6, 8, 10\}$, we also encoded buggy versions thereof (denoted, "buggy-full" and "buggy-simple-only", respectively). Overall, we created 100 configurations of pairs of algorithm encoding with n processes.

To address RQ3, we began by generating 25 linearizable histories of length $l \in \{200, 500, 1000\}$ with $n \in \{5, 8, 11, 14, 17, 20\}$ processes, by randomly executing an atomic snapshot implementation, and recording its actions. We then generated 25 non-linearizable histories of length $l \in \{50, 100, 200\}$ with $n \in \{3, 4, 5, 6, 8, 10\}$ processes as follows: we generated a random linearizable history, and changed its 20-length suffix by randomly changing the values of the scan responses. We repeated this process until we obtained 25 non-linearizable histories. In the context of RQ3, we refer to a choice of l, n, and "linearizable/non-linearizable" as a configuration. This resulted in 900 histories, divided into 18 linearizable and 18 non-linearizable configurations, added to our corpus.

7.5 Experiments and Results

In this section, we detail the experiments we performed to tackle our research questions, and report our results. All experiments were performed on a rather ordinary laptop with an Intel Core i7-6820HK CPU and 32GB of DDR4 RAM, running Windows 10 21H1 and the WSL2 Ubuntu 20.04.2 image from Microsoft.

Table 1 Results of bug detection in non-linearizable implementations. b: max bound on #operation per process, t: time used (sec.), and s: memory used (GB).

test	normal			simple			polynomial			normal			simple			polynomial		
	b	t	s	b	t	s	b	t	s	b	t	s	b	t	s	b	t	s
algorithm	3 processes									4 processes								
BOWMAN	∞	11	0.64	∞	2	0.1	∞	2	0.3	3	168	12.0	∞	60	0.5	∞	54	0.6
JAYANTI	14	272	22.8	23	393	25.1	3	72	0.7	3	149	11.9	6	218	16.0	3	98	0.7
AFEK1	-	-	-	2	242	1.9	2	417	1.8	-	-	-	2	305	2.0	2	506	2.3
AFEK2	2	50	0.5	4	422	4.3	3	215	1.4	2	163	2.0	4	522	7.0	3	255	2.1
RIANY	6	285	4.1	9	445	5.4	27	595	2.0	3	172	12.1	6	275	17	24	512	1.9
algorithm	5 processes									6 processes								
BOWMAN	1	9	0.4	3	290	18.4	40	575	4.4	1	66	3.8	1	7	0.5	35	507	3.3
JAYANTI	1	141	0.9	3	253	18.4	3	118	0.7	1	302	4.6	2	590	28.1	3	142	0.7
AFEK1	-	-	-	2	417	4.0	-	-	-	-	-	-	-	-	-	-	-	-
AFEK2	1	21	0.4	3	441	18.4	3	298	2.6	1	95	4.2	1	7	0.5	3	326	2.8
RIANY	-	-	-	-	-	-	23	579	1.8	-	-	-	-	-	-	21	508	2.2
algorithm	8 processes									10 processes								
BOWMAN	-	-	-	1	288	15.6	33	554	2.7	-	-	-	-	-	-	30	558	2.9
JAYANTI	-	-	-	1	386	15.6	3	197	0.7	-	-	-	-	-	-	3	267	1.1
AFEK1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
AFEK2	-	-	-	1	256	15.7	3	402	3.2	-	-	-	-	-	-	3	491	3.6
RIANY	-	-	-	-	-	-	19	502	2.2	-	-	-	-	-	-	18	586	3.0

7.5.1 Verification Experiments

To address RQ1 and RQ2, we used PAT to verify the correctness of our configurations. We ran three different types of linearizability experiments:

normal. Using PAT's standard linearizability checker with full and buggy-full configurations.

simple. Using PAT's standard linearizability checker with simple-only and buggy-simple-only configurations.

polynomial. Using PAT's reachability checker with simple-only and buggy-simple-only configurations, in parallel to the automaton threads that detect bugs.

Furthermore, for each configuration and matching experiment type, we limited the number of operations that each process was allowed to perform. As some algorithms employ infinite data types (e.g. integers), at least in those cases, the bound is mandatory for PAT to terminate. We set a timeout of 10 min. for buggy implementations, and 1 hr. for correct implementations. We repeated each experiment with various bounds until we found the maximal bound for which each experiment terminated in the allotted time.

▶ Remark 19. For simple configurations, we checked only $(0, 1)$-simple histories. For full verification, it is required to test all (i, j)-histories. Nevertheless, this observation does not affect the feasibility of the approach, since the tests for (i_0, j_0) and (i_1, j_1) simple histories are independent, and can even run on separate machines. Furthermore, symmetry arguments may increase confidence even when checking only $(0, 1)$-simple histories.

▶ Remark 20. We also tried to use Cave [21, 50] and its extension Poling [45, 55], static analysis-based linearizability verifiers. Unfortunately, despite our best efforts, we could not make either tool work for the algorithms we tried to encode. Even for toy correct and

Table 2 Results of verification of linearizable implementations. b: max bound on #operation per process, t: time used (sec.), and s: memory used (GB).

test	normal			simple			polynomial			normal			simple			polynomial		
	b	t	s	b	t	s	b	t	s	b	t	s	b	t	s	b	t	s
algorithm	3 processes									4 processes								
BOWMAN	2	47	0.7	∞	3	0.1	∞	11	0.1	1	29	0.6	∞	134	1.8	∞	460	0.7
JAYANTI	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
AFEK1	1	113	1.0	1	15	0.2	1	31	0.3	-	-	-	-	-	-	-	-	-
AFEK2	1	12	0.2	2	2627	19.5	2	2675	7.5	-	-	-	1	154	1.9	1	269	1.5
RIANY	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
algorithm	5 processes									6 processes								
BOWMAN	-	-	-	1	158	1.4	1	121	0.5	-	-	-	-	-	-	-	-	-
JAYANTI	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
AFEK1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
AFEK2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
RIANY	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

incorrect algorithms that operate atomically, Cave reported errors, and Poling returned unexpected responses. Perhaps the snapshot object deviates from the types of data structures that these tools aim to handle. To the best of our knowledge, PAT, Cave, and Poling are the only available tools that verify linearizability automatically, without requiring additional input.

We present the bug-detection results in Table 1, and the verification results in Table 2. For each configuration and linearizability experiment type, we report the maximal bound on the number of operations per process, for which the experiment terminated before the timeout. If the experiment terminated without imposing a bound, we report the value ∞. Furthermore, for the max bound we found, we report time and space consumption by the corresponding linearizability experiment. As an example, for RIANY buggy-simple-only with 4 processes, when we ran the polynomial linearizability experiment, we found the bug while limiting each process to 24 operations. The execution took 512 sec. and consumed 1.9 GB. Accordingly, in the upper part of Table 1, the cells on the row titled "RIANY" and the columns titled "polynomial; 4 processes" read: b:24, t:512, and s:1.9.

7.5.2 Testing Experiments

To address RQ3, we tested all linearizable and non-linearizable generated histories, applying two methods: our implemented method, and a tool by Lowe [37, 41], with a 10 min. timeout. For each configuration and each tool, we report the percentage of tests that successfully terminated within the allotted time. Furthermore, for the terminated executions, we report the median running time and space consumption. Table 3 presents results for linearizable configurations, and Table 4 for non-linearizable configurations. As an example, when we applied our method to linearizable histories of length 500 with 20 processes, 100% of the tests were successful, the median running time was 0.15sec, and the median space consumption was 150MB. Hence, in Table 3, the cells on the rows titled "500:terminated", "500;median time", and "500;median space" with the column titled "20;This paper" read 100%, 0.15, and 150, respectively.

Table 3 Linearizable simple history testing results. Terminated tests (%), median time used (sec.), and median memory used (MB).

#processes		5		8		11		14		17		20	
len. \ mth.		This paper	Lowe	This paper	Lowe	This paper	Lowe	This paper	Lowe	This paper	Lowe	This paper	Lowe
200	term.	100%	100%	100%	100%	100%	100%	100%	84%	100%	52%	100%	32%
	time	0.02	0.20	0.02	0.22	0.02	0.28	0.02	1.34	0.02	1.11	0.03	2.08
	space	130	336	131	352	131	360	131	456	131	444	132	1228
500	term.	100%	100%	100%	100%	100%	100%	100%	80%	100%	48%	100%	16%
	time	0.04	0.20	0.06	0.22	0.09	0.30	0.12	1.45	0.16	12.57	0.15	0.21
	space	132	216	136	336	140	352	144	492	150	2414	150	346
1000	term.	100%	100%	100%	100%	100%	100%	100%	72%	100%	44%	100%	16%
	time	0.07	0.21	0.14	0.21	0.21	0.33	0.33	1.36	0.40	3.29	0.56	0.85
	space	136	344	146	336	156	352	171	482	182	1620	203	398

Table 4 Non-linearizable simple history testing results. Terminated tests (%), median time used (sec.), and median memory used (MB).

#processes		3		4		5		6		8		10	
len. \ mth.		This paper	Lowe	This paper	Lowe	This paper	Lowe	This paper	Lowe	This paper	Lowe	This paper	Lowe
50	term.	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
	time	0.06	0.28	0.06	0.32	0.06	0.34	0.06	0.35	0.06	0.65	0.06	1.19
	space	129	336	129	348	129	336	129	344	129	368	129	440
100	term.	100%	100%	100%	100%	100%	84%	100%	52%	100%	32%	100%	8%
	time	0.03	0.48	0.03	1.45	0.03	3.76	0.03	24.61	0.03	61.93	0.03	179.80
	space	129	340	129	508	129	2784	129	17896	129	12996	129	12192
200	term.	100%	36%	100%	4%	100%	0%	100%	0%	100%	0%	100%	0%
	time	0.03	30.78	0.03	5.11	0.03	-	0.03	-	0.03	-	0.03	-
	space	129	20392	129	3216	129	-	129	-	129	-	129	-

7.6 Analysis of the Results

Focusing on simple histories is beneficial, as both simple and polynomial outperform the normal linearizability method of PAT for finding bugs. In addition, overall, polynomial performs better than simple (see Table 1). normal found the bug in 14/30 cases, with up to 6 processes. simple succeeded in 3 additional cases with up to 8 processes, with polynomial succeeding in 26/30 cases with up to 10 processes. Importantly, simple allows for larger bounds than normal in 16/17 cases, and the same bound in the remaining case. polynomial allows for larger bounds than simple in 16 cases, and smaller bounds in 5 cases. This indicates that polynomial enables deeper exploration than simple, and thus we conclude that it is more efficient. Both polynomial and simple manage to explore implementations significantly deeper than normal. We also observe that polynomial consumes less space, which is the main bottleneck of model checking, than simple. The peak memory consumption we recorded for polynomial was 4.4GB, whereas the peak we recorded for simple was 25.1GB, and 9/20 executions with more than 10GB. While the peak we recorded for normal was 22.8GB, it scaled much worse than simple and failed to cope with the more challenging configurations.

In Table 2, we see that simple and polynomial enable the verification of BOWMAN with 3 and 4 processes. To the best of our knowledge, this is the first time that this algorithm has been verified to some extent. Model-checking techniques are complete and mainly efficient for

bug detection. Verifying a concurrent algorithm for 4 processes is noteworthy (compare, e.g., to the results of [39]). Yet, excluding these results, although both methods perform better than normal, we did not manage to verify other implementations. We mention that, in some old evaluations we performed, we used a prototype tool we wrote that uses simple histories (but does not employ the automatons of Section 4.1), and managed to verify JAYANTI with 3 processes within 21 sec. (reference hidden for double-blind review). As this deviates from what Table 2 illustrates, we believe that further investigation is required.

Tables 3 and 4 show that our testing technique outperforms [41] by several orders of magnitude, mainly and most importantly, in terms of feasibility. Our tool easily handled all 900 histories, while the competitor failed to cope with challenging configurations, successfully handling only 554/900 histories. We also observe that our technique is scalable. The differences in time and space consumption between extremum values are negligible.

Moreover, we note that our technique is insensitive to the correctness of the tested history. In contrast, our competitor quickly fails over non-linearizable histories. To gain more confidence in this observation, we further generated 25 non-linearizable histories of length 1000 for 20 processes, with a linearizable prefix of length at least 980. Our tool handled all with a median running time of 0.55sec. Note that our competitor failed almost entirely over non-linearizable histories of length 200, with 3-10 processes.

8 Related Work

Alur et al. proposed an EXPSPACE-technique for verifying linearizability [5], and Hamza proved EXPSPACE-completeness [29]. Bouajjani et al. proved the undecidability of linearizability of infinite-state systems, and the PSPACE-completeness of linearizability with fixed linearization points [14].

Due to the high complexity of the problem, sound and complete model-checking techniques manage to perform limited verification with up to 3 processes [19, 39, 52]. [39] also verifies a stack implementation for 4 processes, but only by limiting the stack size to two data values. Hsu et al. [33] proposed a bounded model checking technique for hyper-LTL, and used it to rediscover known bugs (see [25]) in the “Snark” dequeue implementation [24].

Static analysis efforts are incomplete, but can work for infinite-state implementations. However, most ask for additional information from the user. The works [3, 6, 13, 50] ask for linearization points, some in a conditional manner. The work [2] asks for linearization policies. The work [47] asks for the specification of sub-operations and relations between them. Cave [21, 51] and Poling [55] work without further information. However, as we report in Section 7, we did not manage to work with these tools. Perhaps the snapshot object deviates from the types of data structures that these tools aim to handle.

The way we employ the data independence property resembles Abdulla et al. [1]. They ran automatons in parallel to queue and stack implementations to detect bugs. Their approach is incomplete, but works for infinite-state implementations. However, their automatons detect incorrect sequential histories, in contrast to concurrent histories as we do, and thus their approach requires specifying linearization points. It is rather simple to construct an automaton that detects incorrect sequential snapshot histories, hence their approach can be applied to the snapshot object straightforwardly. But, as linearization points of snapshot implementations are evasive, the benefit of doing so is questionable.

Other works also focused on specific data structures. Bouajanni et al. [15] prove that verification of data-independent queue, stack, register, and mutex implementations is PSPACE-complete for a fixed number of processes, and EXPSPACE-complete for infinitely many

processes. In [16], Bouajjani et al. extend the latter result to data-independent and projection-closed priority queues. To the best of our knowledge, those techniques have not been implemented or evaluated. Chakraborty et al. [22] identified conditions that are equivalent to the linearizability of data-independent queue implementations, and use them to automatically verify Herlihy and Wing's queue [31]. Abdulla et al. [3] used those conditions and the results of [22] to extend their static analysis technique [2] to verify stack and queue implementations without linearization points.

Wing and Gong considered the problem of testing linearizability, and gave an exponential-time algorithm [53]. Gibbons and Korach proved NP-completeness [28], and further showed that register-histories with k processes can be tested in time $O(n2^{O(k)}+n\log n)$. Lowe [41] suggested optimizations for the algorithm of [53]. Horn and Kroening suggested an optimization that applies to set implementations [32]. Emmi and Enea [27] identified a class of data structures for which a polynomial-time testing algorithm exists. This class includes queue, stack, set, and map, but does not include snapshot.

9 Conclusion

We proved that a data-independent snapshot algorithm is linearizable if and only if all of its simple histories are linearizable. This gives rise to an optimization for proving/disproving the correctness of snapshot implementations, i.e, examining only simple histories. This optimization can exponentially reduce the number of reachable states to inspect. Moreover, we proved that non-linearizable simple histories are identified by a polynomial-sized automaton. This enables a polynomial-time technique for verifying the linearizability of snapshot implementations, and a linear-time technique for testing the linearizability of snapshot histories. We implemented our techniques, and reported on evaluations that support the efficiency of our methods over existing techniques.

Future Work

We wonder if the notion of simple histories can be replicated to other data structures. In particular, it would be interesting to investigate whether such an adaptation would admit automata-based verification/testing techniques similar to those we presented for the snapshot object. The automatons presented in [14] seem like a good place to begin in order to define simple histories geared at queues and stacks. Another future direction is to extend our results to multi-writer snapshots, and to implementations that are strongly linearizable [43].

References

1 Parosh Aziz Abdulla, Frédéric Haziza, Lukás Holík, Bengt Jonsson, and Ahmed Rezine. An integrated specification and verification technique for highly concurrent data structures for highly concurrent data structures. *Int. J. Softw. Tools Technol. Transf.*, 19(5):549–563, 2017. doi:10.1007/s10009-016-0415-4.

2 Parosh Aziz Abdulla, Bengt Jonsson, and Cong Quy Trinh. Automated verification of linearization policies. In Xavier Rival, editor, *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, volume 9837 of *Lecture Notes in Computer Science*, pages 61–83. Springer, 2016. doi:10.1007/978-3-662-53413-7_4.

3 Parosh Aziz Abdulla, Bengt Jonsson, and Cong Quy Trinh. Fragment abstraction for concurrent shape analysis. In Amal Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 442–471. Springer, 2018. doi:10.1007/978-3-319-89884-1_16.

4 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993. doi:10.1145/153724.153741.

5 Rajeev Alur, Kenneth L. McMillan, and Doron A. Peled. Model-checking of correctness conditions for concurrent objects. *Inf. Comput.*, 160(1-2):167–188, 2000. doi:10.1006/inco.1999.2847.

6 Daphna Amit, Noam Rinetzky, Thomas W. Reps, Mooly Sagiv, and Eran Yahav. Comparison under abstraction for verifying linearizability. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 477–490. Springer, 2007. doi:10.1007/978-3-540-73368-3_49.

7 James H. Anderson. Composite registers. *Distributed Comput.*, 6(3):141–154, 1993. doi:10.1007/BF02242703.

8 James H. Anderson. Multi-writer composite registers. *Distributed Comput.*, 7(4):175–195, 1994. doi:10.1007/BF02280833.

9 James Aspnes and Maurice Herlihy. Wait-free data structures in the asynchronous PRAM model. In Frank Thomson Leighton, editor, *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '90, Island of Crete, Greece, July 2-6, 1990*, pages 340–349. ACM, 1990. doi:10.1145/97444.97701.

10 Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Efficient atomic snapshots using lattice agreement (extended abstract). In Adrian Segall and Shmuel Zaks, editors, *Distributed Algorithms, 6th International Workshop, WDAG '92, Haifa, Israel, November 2-4, 1992, Proceedings*, volume 647 of *Lecture Notes in Computer Science*, pages 35–53. Springer, 1992. doi:10.1007/3-540-56188-9_3.

11 Hagit Attiya and Ophir Rachman. Atomic snapshots in o(n log n) operations. *SIAM J. Comput.*, 27(2):319–340, 1998. doi:10.1137/S0097539795279463.

12 Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT Press, 2008.

13 Josh Berdine, Tal Lev-Ami, Roman Manevich, G. Ramalingam, and Shmuel Sagiv. Thread quantification for concurrent shape analysis. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 399–413. Springer, 2008. doi:10.1007/978-3-540-70545-1_37.

14 Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. Verifying concurrent programs against sequential specifications. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 290–309. Springer, 2013. doi:10.1007/978-3-642-37036-6_17.

15 Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. On reducing linearizability to state reachability. *Inf. Comput.*, 261:383–400, 2018. doi:10.1016/j.ic.2018.02.014.

16 Ahmed Bouajjani, Constantin Enea, and Chao Wang. Checking linearizability of concurrent priority queues. In Roland Meyer and Uwe Nestmann, editors, *28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany*, volume 85 of *LIPIcs*, pages 16:1–16:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CONCUR.2017.16.

17 Jack R Bowman. Obstruction-free snapshot, obstruction-free consensus, and fetch-and-add modulo k. Technical report, Technical Report TR2011-681, Dartmouth College, Computer Science, Hanover, NH, 2011.

18 Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 12–21. ACM, 2007. doi:10.1145/1250734.1250737.

19 Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: a complete and automatic linearizability checker. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 330–340. ACM, 2010. doi:10.1145/1806596.1806634.

20 Jacob Burnim, George C. Necula, and Koushik Sen. Specifying and checking semantic atomicity for multithreaded programs. In Rajiv Gupta and Todd C. Mowry, editors, *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, pages 79–90. ACM, 2011. doi:10.1145/1950365.1950377.

21 CAVE Website. https://people.mpi-sws.org/~viktor/cave/. Last Accessed: Aug. 31, 2021.

22 Soham Chakraborty, Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. Aspect-oriented linearizability proofs. *Log. Methods Comput. Sci.*, 11(1), 2015. doi:10.2168/LMCS-11(1:20)2015.

23 Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model checking and the state explosion problem. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2011. doi:10.1007/978-3-642-35746-6_1.

24 David Detlefs, Christine H. Flood, Alex Garthwaite, Paul Alan Martin, Nir Shavit, and Guy L. Steele Jr. Even better dcas-based concurrent deques. In Maurice Herlihy, editor, *Distributed Computing, 14th International Conference, DISC 2000, Toledo, Spain, October 4-6, 2000, Proceedings*, volume 1914 of *Lecture Notes in Computer Science*, pages 59–73. Springer, 2000. doi:10.1007/3-540-40026-5_4.

25 Simon Doherty, David Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul Alan Martin, Mark Moir, Nir Shavit, and Guy L. Steele Jr. DCAS is not a silver bullet for nonblocking algorithm design. In Phillip B. Gibbons and Micah Adler, editors, *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain*, pages 216–224. ACM, 2004. doi:10.1145/1007912.1007945.

26 Cynthia Dwork, Maurice Herlihy, Serge A. Plotkin, and Orli Waarts. Time-lapse snapshots. *SIAM J. Comput.*, 28(5):1848–1874, 1999. doi:10.1137/S0097539793243685.

27 Michael Emmi and Constantin Enea. Sound, complete, and tractable linearizability monitoring for concurrent collections. *Proc. ACM Program. Lang.*, 2(POPL):25:1–25:27, 2018. doi:10.1145/3158113.

28 Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM J. Comput.*, 26(4):1208–1244, 1997. doi:10.1137/S0097539794279614.

29 Jad Hamza. On the complexity of linearizability. *Comput.*, 101(9):1227–1240, 2019. doi:10.1007/s00607-018-0596-7.

30 Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems (ICDCS 2003), 19-22 May 2003, Providence, RI, USA*, pages 522–529. IEEE Computer Society, 2003. doi:10.1109/ICDCS.2003.1203503.

31 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

32 Alex Horn and Daniel Kroening. Faster linearizability checking via p-compositionality. In Susanne Graf and Mahesh Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*, volume 9039 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2015. doi:10.1007/978-3-319-19195-9_4.

33 Tzu-Han Hsu, César Sánchez, and Borzoo Bonakdarpour. Bounded model checking for hyperproperties. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I*, volume 12651 of *Lecture Notes in Computer Science*, pages 94–112. Springer, 2021. doi:10.1007/978-3-030-72016-2_6.

34 Prasad Jayanti. *f*-arrays: implementation and applications. In Aleta Ricciardi, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing, PODC 2002, Monterey, California, USA, July 21-24, 2002*, pages 270–279. ACM, 2002. doi:10.1145/571825.571875.

35 Prasad Jayanti. An optimal multi-writer snapshot algorithm. In Harold N. Gabow and Ronald Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 723–732. ACM, 2005. doi:10.1145/1060590.1060697.

36 Lefteris M. Kirousis, Paul G. Spirakis, and Philippas Tsigas. Reading many variables in one atomic operation: Solutions with linear or sublinear complexity. *IEEE Trans. Parallel Distrib. Syst.*, 5(7):688–696, 1994. doi:10.1109/71.296315.

37 Linearizability Tester Website. http://www.cs.ox.ac.uk/people/gavin.lowe/LinearizabiltyTesting/. Last Accessed: Aug. 31, 2021.

38 Yang Liu, Wei Chen, Yanhong A. Liu, and Jun Sun. Model checking linearizability via refinement. In Ana Cavalcanti and Dennis Dams, editors, *Proceedings of the Second World Congress on Formal Methods (FM'09)*, volume 5850 of *Lecture Notes in Computer Science*, pages 321–337. Springer, 2009.

39 Yang Liu, Wei Chen, Yanhong A. Liu, and Jun Sun. Model checking linearizability via refinement. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5850 of *Lecture Notes in Computer Science*, pages 321–337. Springer, 2009. doi:10.1007/978-3-642-05089-3_21.

40 Yang Liu, Wei Chen, Yanhong A. Liu, Jun Sun, Shao Jie Zhang, and Jin Song Dong. Verifying linearizability via optimized refinement checking. *IEEE Trans. Software Eng.*, 39(7):1018–1039, 2013. doi:10.1109/TSE.2012.82.

41 Gavin Lowe. Testing for linearizability. *Concurr. Comput. Pract. Exp.*, 29(4), 2017. doi:10.1002/cpe.3928.

42 Maged M Michael and Michael L Scott. Correction of a memory management method for lock-free data structures. Technical report, University of Rochester, Computer Science, 1995.

43 Sean Ovens and Philipp Woelfel. Strongly linearizable implementations of snapshots and other types. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 197–206. ACM, 2019. doi:10.1145/3293611.3331632.

44 PAT Website. https://pat.comp.nus.edu.sg/. Last Accessed: Aug. 31, 2021.

45 Poling Website. https://github.com/rowangithub/Poling/. Last Accessed: Aug. 31, 2021.

46 Yaron Riany, Nir Shavit, and Dan Touitou. Towards a practical snapshot algorithm. *Theor. Comput. Sci.*, 269(1-2):163–201, 2001. doi:10.1016/S0304-3975(00)00412-6.

47 Vineet Singh, Iulian Neamtiu, and Rajiv Gupta. Proving concurrent data structures linearizable. In *27th IEEE International Symposium on Software Reliability Engineering, ISSRE 2016, Ottawa, ON, Canada, October 23-27, 2016*, pages 230–240. IEEE Computer Society, 2016. doi:10.1109/ISSRE.2016.31.

48 Jun Sun, Yang Liu, and Bin Cheng. Model checking a model checker: A code contract combined approach. In Jin Song Dong and Huibiao Zhu, editors, *Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings*, volume 6447 of *Lecture Notes in Computer Science*, pages 518–533. Springer, 2010. doi:10.1007/978-3-642-16901-4_34.

49 Supporting materials. https://github.com/hayounav/Thesis_experiments/tree/main/snapshot%20verification%20and%20testing.

50 Viktor Vafeiadis. Shape-value abstraction for verifying linearizability. In Neil D. Jones and Markus Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, volume 5403 of *Lecture Notes in Computer Science*, pages 335–348. Springer, 2009. doi:10.1007/978-3-540-93900-9_27.

51 Viktor Vafeiadis. Automatically proving linearizability. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 450–464. Springer, 2010. doi:10.1007/978-3-642-14295-6_40.

52 Martin T. Vechev and Eran Yahav. Deriving linearizable fine-grained concurrent objects. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 125–135. ACM, 2008. doi:10.1145/1375581.1375598.

53 Jeannette M. Wing and Chun Gong. Testing and verifying concurrent objects. *J. Parallel Distributed Comput.*, 17(1-2):164–182, 1993. doi:10.1006/jpdc.1993.1015.

54 Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*, pages 184–193. ACM Press, 1986. doi:10.1145/512644.512661.

55 He Zhu, Gustavo Petri, and Suresh Jagannathan. Poling: SMT aided linearizability proofs. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 3–19. Springer, 2015. doi:10.1007/978-3-319-21668-3_1.

Almost Universally Optimal Distributed Laplacian Solvers via Low-Congestion Shortcuts*

Ioannis Anagnostides ✉
Carnegie Mellon University, Pittsburgh, PA, USA

Christoph Lenzen ✉
CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

Bernhard Haeupler ✉
ETH Zürich, Switzerland

Goran Zuzic ✉
ETH Zürich, Switzerland

Themis Gouleakis ✉
National University of Singapore, Singapore

Abstract

In this paper, we refine the (almost) *existentially optimal* distributed Laplacian solver recently developed by Forster, Goranci, Liu, Peng, Sun, and Ye (FOCS '21) into an (almost) *universally optimal* distributed Laplacian solver.

Specifically, when the topology is known (i.e., the Supported-CONGEST model), we show that any Laplacian system on an n-node graph with *shortcut quality* $\mathrm{SQ}(G)$ can be solved after $n^{o(1)}\mathrm{SQ}(G)\log(1/\varepsilon)$ rounds, where ε is the required accuracy. This almost matches our lower bound that guarantees that any correct algorithm on G requires $\widetilde{\Omega}(\mathrm{SQ}(G))$ rounds, even for a crude solution with $\varepsilon \leq 1/2$. Several important implications hold in the unknown-topology (i.e., standard CONGEST) case: for excluded-minor graphs we get an almost universally optimal algorithm that terminates in $D \cdot n^{o(1)}\log(1/\varepsilon)$ rounds, where D is the hop-diameter of the network; as well as $n^{o(1)}\log(1/\varepsilon)$-round algorithms for the case of $\mathrm{SQ}(G) \leq n^{o(1)}$, which holds for most networks of interest. Conditioned on improvements in state-of-the-art constructions of low-congestion shortcuts, the CONGEST results will match the Supported-CONGEST ones.

Moreover, following a recent line of work in distributed algorithms, we consider a hybrid communication model which enhances CONGEST with limited global power in the form of the node-capacitated clique (NCC) model. In this model, we show the existence of a Laplacian solver with round complexity $n^{o(1)}\log(1/\varepsilon)$.

The unifying thread of these results, and our main technical contribution, is the study of a novel ρ-*congested* generalization of the standard *part-wise aggregation* problem. We develop near-optimal algorithms for this primitive in the Supported-CONGEST model, almost-optimal algorithms in (standard) CONGEST (with the additional overhead due to standard barriers), as well as a simple algorithm for bounded-treewidth graphs with a quadratic dependence on the congestion ρ. This primitive can be readily used to accelerate the Laplacian solver of Forster, Goranci, Liu, Peng, Sun, and Ye, and we believe it will find further independent applications in the future.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed algorithms, Laplacian solvers, low-congestion shortcuts

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.6

Related Version *Full Version*: https://arxiv.org/abs/2109.05151

* The author ordering was randomized using https://www.aeaweb.org/journals/policies/random-author-order/generator. It is requested that citations of this work list the authors separated by \textcircled{r} instead of commas: Anagnostides ⓡ Lenzen ⓡ Haeupler ⓡ Zuzic ⓡ Gouleakis.

© Ioannis Anagnostides, Christoph Lenzen, Bernhard Haeupler, Goran Zuzic, and Themis Gouleakis; licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 6; pp. 6:1–6:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Funding *Bernhard Haeupler*: Supported in part by NSF grants CCF-1814603, CCF-1910588, NSF CAREER award CCF-1750808, a Sloan Research Fellowship, funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (ERC grant agreement 949272), and the Swiss National Foundation (project grant 200021-184735).
Goran Zuzic: Supported in part by the Swiss National Foundation (project grant 200021-184735).
Themis Gouleakis: Supported in part by an NRF Fellowship for AI (R-252-000-A33-133). Part of the work was done while visiting the Simons Institute for Theory of Computing).

Acknowledgements We are grateful to the anonymous reviewers for their valuable feedback.

1 Introduction

The *Laplacian paradigm* has emerged as one of the cornerstones of modern algorithmic graph theory. Integrating techniques from combinatorial optimization with powerful machinery from numerical linear algebra, it was originally pioneered in [47] who established the first nearly-linear time solvers for a (linear) Laplacian system. Thereafter, there has been a considerable amount of interest in providing simpler and more efficient solvers [34, 33, 37]. Indeed, this framework has led to some state of the art algorithms for a wide range of fundamental graph-theoretic problems; e.g., see [5, 40, 10, 48, 32, 43, 4], and references therein. In the distributed setting, a major breakthrough was very recently made in [18]. In particular, the authors developed a distributed algorithm that solves any Laplacian system on an n-node graph after $n^{o(1)}(\sqrt{n} + D)\log(1/\varepsilon)$ rounds of the standard CONGEST model, where D represents the hop-diameter of the underlying network and $\varepsilon > 0$ is the error of the solver. Moreover, they showed that their algorithm is *existentially optimal*, up to the $n^{o(1)}$ factor, establishing a lower bound of $\widetilde{\Omega}(\sqrt{n} + D)$ rounds via a reduction from the $s - t$ connectivity problem [13].

This *existential* lower bound in the CONGEST model of distributed computing should hardly come as any surprise. Indeed, it is well-known by now that a remarkably wide range of *global* optimization problems, including minimum spanning tree (MST), minimum cut (Min-Cut), maximum flow, and single-source shortest paths (SSSP), require $\widetilde{\Omega}(\sqrt{n} + D)$ rounds[1] [41, 15, 13]. The same limitation generally applies to any non-trivial approximation and even under randomization. Nonetheless, these lower bounds are constructed on some pathological graph instances that arguably do not occur in practice. This begs the question: *Can we obtain more refined performance guarantees based on the underlying topology of the communication network?* The framework of *low-congestion shortcuts*, introduced by [20], demonstrated that bypassing the notorious $\Omega(\sqrt{n})$ lower bound is possible: MST and Min-Cut on *planar graphs* can be solved in $\widetilde{O}(D)$ rounds. This is crucial, given that in many graphs of practical significance the diameter is remarkably small; e.g., $D = \text{polylog}(n)$ (as is folklore, this holds for most social networks), implying *exponential improvements* over generic algorithms used for general graphs. In the context of the distributed Laplacian paradigm, we raise the following question:

Is there a faster distributed Laplacian solver under "non-worst-case" families of graphs in the CONGEST model?

The only known technique in distributed computing for designing algorithms that go below the $\sqrt{n}$-bound is the low-congestion shortcut framework of Ghaffari and Haeupler [20], and the large ecosystem of tools built around it [25, 26, 29, 21, 50, 23, 27]. However, the "ρ-congested minor" primitive introduced and extensively used in the novel distributed Laplacian

[1] As usual, we use the notation $\widetilde{O}(\cdot)$ and $\widetilde{\Omega}(\cdot)$ to suppress polylogarithmic factors on n.

solver [18] is out of reach from the current set of tools available in the low-congestion shortcut framework. We address this issue by introducing an analogous primitive called *ρ-congested part-wise aggregation*, which greatly simplifies the interface used by [18]. We then extend the low-congestion shortcut framework with new techniques that enables it to near-optimally solve this primitive: we provide both an algorithm that utilizes the very recent hop-constrained expander decompositions for shortcut construction [27] to solve the primitive in general graphs with a linear dependence on ρ, as well as a very simple algorithm with a quadratic ρ-dependence for bounded-treewidth graphs. Finally, we settle our original question in the positive by establishing that our new primitive can be readily used to accelerate the distributed Laplacian solver for non-worst-case topologies.

Specifically, we show our new techniques are sufficient to lift the existentially optimal algorithm [18] to a *universally optimal* algorithm – modulo $n^{o(1)}$ factor inherent in the prior approach – for distributedly solving a Laplacian system, meaning that, *for any topology*, our algorithm is essentially as fast as possible. In other words, for any graph, our algorithm almost matches the best possible (correct) algorithm for that graph. This result is unconditional in essentially all settings of interest (see Theorem 2 for details), but relies on conjectured improvements of current state-of-the-art constructions of low-congestion shortcuts to achieve unqualified universal optimality – like all other results in the area.

Furthermore, another concrete way of bypassing the $\widetilde{\Omega}(\sqrt{n} + D)$ lower bound, besides investigating non-worst-case families of graphs, is by enhancing the local communication network with a limited amount of *global power*. Indeed, research concerning *hybrid* networks was recently initiated in the realm of distributed algorithms [3], although networks combining different communication modes have already found numerous applications in real-life computing systems; as such, hybrid networks have been intensely studied in other areas of distributed computing (see [9, 49, 31], and references therein). In this paper, we will enhance the standard CONGEST model with the recently introduced *node-capacitated clique* (henceforth NCC) [2]. The latter model enables all-to-all communication, but with severe capacity restrictions for every node. The integration of these models will be referred to as the HYBRID model for the rest of this work. This leads to the following central question:

Is there a faster distributed Laplacian solver in the HYBRID model?

Our paper essentially settles this question by showing the same ρ-congested part-wise aggregation primitive can be efficiently solved in $\widetilde{O}(\rho)$ rounds of NCC, implying an almost optimal $n^{o(1)}$-round distributed algorithm for solving Laplacian systems in the HYBRID model. A conceptual contribution of our approach is that we treat both CONGEST, Supported-CONGEST, and HYBRID in a *unified way* through the lens of the low-congestion shortcut framework, by designing our algorithm using high-level primitives and leaving the model-specific translations to the framework itself. A similar unified view of PRAM (i.e., parallel) and CONGEST (i.e., distributed) graph algorithms through the same lens has led to very recent breakthroughs on long-standing open problems for both of these settings [45].

1.1 Overview of our Contributions and Techniques

The unifying thread and the main technical ingredient of our (almost) universally optimal distributed Laplacian solvers is a new fundamental communication primitive referred to as the *congested part-wise aggregation problem.* Specifically, we develop near-optimal algorithms for solving this problem in the (Supported-)CONGEST and the NCC model (Section 3), and then we utilize this primitive to develop almost universally optimal Laplacian solvers.

1.1.1 The Congested Part-Wise Aggregation Problem

To introduce the congested part-wise aggregation problem, let us first give some basic background. The aforementioned Ghaffari-Haeupler framework of low-congestion shortcuts revolves around the so-called *part-wise aggregation problem* posed as follows: "The graph is partitioned into *disjoint* and individually-connected parts, and we need to compute some simple aggregate function for each part, e.g., the minimum of the values held by the nodes in a given part" [20] (see Definition 4 for a formal definition). Importantly, it has been shown that this primitive can be solved efficiently in *structured* topologies and that many problems (including the MST, shortest path, min-cut, etc.) reduce to a small number of calls to a part-wise aggregation oracle, leading to universally optimal algorithms. Unfortunately, it is not clear how to reduce solving a Laplacian system to (a small number of) part-wise aggregation calls; in this paper, we primarily address this issue.

Our first technical contribution is to extend the framework of low-congestion shortcuts by studying a more general primitive: one that incorporates *congestion* (of the input parts) into the underlying *part-wise aggregation* instance. More precisely, unlike the standard part-wise aggregation problem, we allow each node to participate in up to $\rho \in \mathbb{Z}_{\geq 1}$ aggregation parts (see Definition 13). We later show that efficient solutions to this primitive leads to efficient distributed Laplacian solvers.

We first remark that a natural strategy for solving congested part-wise aggregation instances does not work: congested instances *cannot*, in general, be directly reduced to a "small" collection of 1-congested instances, thereby necessitating a more refined approach. To this end, our approach is based on "lifting" the underlying communication network $\overline{G}$ into its *ρ-layered version* $\widehat{G}_{O(\rho)}$: every edge is replaced with a matching and every node with a ρ-clique. The importance of this transformation is that, as we show in Lemma 15, the ρ-congested part-wise aggregation problem can be reduced to a 1-congested instance on the ρ-layered graph (Section 3.1.1). This is first established under the assumption that individual parts correspond to simple paths, and then we extend our results to general parts by following [29]. In light of this reduction, we next focus on solving the 1-congested part-wise aggregation instance on the layered graph.

As a warm-up, we treat graphs with bounded *treewidth* $\text{tw}(G)$ (Definition 11). It is known from [26] that on a graph G with treewidth $\text{tw}(G)$, a 1-congested part-wise aggregation instance can be solved in $\widetilde{O}(\text{tw}(G)D)$ rounds of CONGEST. Keeping this in mind, we first show that the treewidth of the ρ-layered graph $\widehat{G}_\rho$ can only increase by a factor of ρ compared to the original graph (Lemma 19). Hence, we can solve 1-congested instances in $\widehat{G}_{O(\rho)}$ in $\widetilde{O}(\rho\, \text{tw}(\overline{G})D)$ rounds (when the underlying network is $\widehat{G}_{O(\rho)}$), which in turn allows us to solve ρ-congested instances on $\overline{G}$ in $\widetilde{O}(\rho^2\, \text{tw}(G)D)$ time in G (another ρ factor is necessary to simulate $\widehat{G}_{O(\rho)}$ in $\overline{G}$). This positive result poses a natural question: can we achieve similar results on graphs with bounded *minor density* $\delta(G)$ (Definition 9)? However, the answer to this question is negative: *minor density* can blow up even for a 2-layered planar graph (see Observation 21), making such a result impossible.

Then, we look at arbitrary graphs G: it is known that 1-congested part-wise aggregation instances can be solved in a number of rounds that is controlled by $\text{SQ}(G)$, where $\text{SQ}(G)$ is the *shortcut quality* of G (a certain graph parameter we formalize in Definition 7). Specifically, it can be solved in $\widetilde{O}(\text{SQ}(G))$ rounds when the topology is known in advance[2] [29] and $\text{poly}(\text{SQ}(G)) \cdot n^{o(1)}$ in general CONGEST [27]. The shortcut quality parameter is significant

[2] This model is also known as the *supported* CONGEST. That is, CONGEST under the assumption that the topology is known; see Section 2 for a formal description of the model. Our techniques also apply in the full generality of CONGEST, as we explain in the sequel.

since many distributed problems (including the MST, shortest path, min-cut, and – Laplacian solving, as we show later) require $\widetilde{\Omega}(\mathrm{SQ}(G))$ rounds in CONGEST to be solved on G [29]. Therefore, algorithms that have an upper bound close to $\mathrm{SQ}(G)$ are *universally optimal*.

With the end goal of solving the 1-congested part-wise aggregations on layered graphs $\widehat{G}_\rho$ in time controlled by $\mathrm{SQ}(G)$, our main result established that *the shortcut quality of the ρ-layered graph does not increase* (modulo polylogarithmic factors) as compared to the original graph (Theorem 22). This has a plethora of important consequences: (1) when $\mathrm{SQ}(G) \leq n^{o(1)}$, we can unconditionally solve ρ-congested part-wise aggregation instances in $\rho \cdot n^{o(1)}$ CONGEST rounds using the state-of-the-art shortcut construction [27], and (2) when the topology of G is known (i.e., Supported-CONGEST), there exists a distributed algorithm that solves any ρ-congested part-wise aggregation problem in $\rho \cdot \widetilde{O}(\mathrm{SQ}(G))$ rounds via [29]. As a consequence of our general result, the shortcut quality of any 2-layered planar graph is $\widetilde{O}(D)$ since the shortcut quality of a planar graph is $\widetilde{O}(D)$ [20]. This is perhaps the most natural example of a graph whose minor density is very far from the shortcut quality; the only other example documented in the literature so far is that of *expander graphs*.

Our proof proceeds by employing alternative characterizations of the shortcut quality in terms of certain communication tasks. Specifically, shortcut quality can be shown to be equal (modulo polylogarithmic factors) to the following two-player max-min game: the first (max) player chooses k sources and k sinks in the graph such that we can find k node-disjoint paths matching the sources with the sinks; then the second (min) player finds the smallest so-called *quality* Q such that there exist k paths matching the sources with the sinks with the path lengths being at most Q and each edge of the underlying graph supporting at most Q of second player's paths. This characterization allows us to compare the shortcut quality of $\widehat{G}_\rho$ with $\overline{G}$ as follows: take the worst-case (first player's) set of sources and sinks in $\widehat{G}_\rho$. Project them to $\overline{G}$ and note they have node congestion ρ (due to the construction of $\widehat{G}_\rho$). Then, we show we can decompose (i.e., partition) these set of sources and sinks into $\widetilde{O}(\rho)$ pairs of sub-sources and sub-sinks that are node-disjointly connectable in G. However, each such set enjoys paths of quality $\mathrm{SQ}(G)$, hence embedding each such pair in a separate layer of $\widehat{G}_\rho$ shows that the shortcut quality of $\mathrm{SQ}(\widehat{G}_\rho)$ is at most $\widetilde{O}(\mathrm{SQ}(\overline{G}))$. Although this general approach improves over our result for treewidth-bounded graphs we previously described, our approach for the latter class of graphs is substantially simpler and more suited in practice.

1.1.2 Almost Universally Optimal Laplacian Solvers

First, we note that any distributed Laplacian solver that always correctly outputs an answer on a fixed graph G must take at least $\tilde{\Omega}(\mathrm{SQ}(G))$ rounds, giving us a lower bound to compare ourselves with. Our refined lower bound uses the hardness result recently shown by [29] for the spanning connected subgraph problem, applicable for *any* (i.e., non-worst-case) graph G. Specifically, we show that a Laplacian solver can be leveraged to solve the spanning connected subgraph problem, thereby substantially strengthening the lower bound in [18].

▶ **Theorem 1.** *Consider a graph $\overline{G}$ with shortcut quality $\mathrm{SQ}(\overline{G})$. Then, solving a Laplacian system on $\overline{G}$ with $\varepsilon \leq \frac{1}{2}$ requires $\widetilde{\Omega}(\mathrm{SQ}(\overline{G}))$ rounds in both CONGEST and Supported-CONGEST models.*

On the upper-bound side, we utilize the congested part-wise aggregation primitive to improve and refine the Laplacian solver of [18], leading to a substantial improvement in the round complexity under *structured* network topologies.

▶ **Theorem 2.** *Consider any n-node graph G with shortcut quality* $\mathrm{SQ}(G)$ *and hop-diameter D. There exists a distributed Laplacian solver with error $\varepsilon > 0$ with the following guarantees:*

- *In the Supported-CONGEST model, it requires* $n^{o(1)}\,\mathrm{SQ}(G)\log(1/\varepsilon)$ *rounds.*
- *In the CONGEST model, it requires* $n^{o(1)}\,\mathrm{poly}(\mathrm{SQ}(G))\log(1/\varepsilon)$ *rounds.*
- *In the CONGEST model on graphs with minor density δ, it requires* $n^{o(1)}\delta D\log(1/\varepsilon)$ *rounds.*

We note that the above algorithm is almost (up to inherent $n^{o(1)}$ factors) universally optimality for most settings of interest. Since it is (almost) matching the $\mathrm{SQ}(G)$-lower-bound, it is unconditionally universally optimal when the topology is known in advance (i.e., Supported-CONGEST). Furthermore, in standard CONGEST, we give almost universally optimal $Dn^{o(1)}\log(1/\varepsilon)$-round algorithms for topologies that include planar graphs, $n^{o(1)}$-*genus* graphs, $n^{o(1)}$-treewidth graphs, excluded-minor graphs, since all of them are graphs with minor density $\delta(G) = n^{o(1)}$. Furthermore, for the realistic case of $D \le n^{o(1)}$, it holds for most networks of interest that $\mathrm{SQ}(G) \le n^{o(1)}$ (e.g., expanders, hop-constrained expanders, as well as all classes mentioned earlier), for which we get $n^{o(1)}\log(1/\varepsilon)$-round solvers. Finally, the conjectured improvements of the state-of-the-art of almost-optimal low-congestion shortcut constructions would immediately lift our results to be unconditionally universally optimal in CONGEST; this issue is orthogonal and not within the scope of this paper.

Furthermore, in HYBRID we obtain an almost optimal complexity in *general graphs*:

▶ **Theorem 3.** *Consider any n-node graph. There exists a distributed Laplacian solver in the HYBRID model with round complexity* $n^{o(1)}\log(1/\varepsilon)$*, where $\varepsilon > 0$ is the error of the solver.*

This implies a remarkably fast subroutine for solving a Laplacian system in HYBRID under arbitrary topologies. As a result, we corroborate the observation that a very limited amount of global power can lead to substantially faster algorithms for certain optimization problems, supplementing a recent line of work [8, 3, 35, 16, 7, 24, 36, 11]. Furthermore, our framework based on the congested part-wise aggregation problem allows for a unifying treatment of both (Supported-)CONGEST and HYBRID, and we consider this to be an important conceptual contribution of our work. Indeed, as we previously explained, both of our accelerated Laplacian solvers rely on faster algorithms for solving the congested part-wise aggregation problem. In particular, for (Supported-)CONGEST we have already described our approach in detail, while in the HYBRID model we employ certain communication primitives developed in [2] for dealing with congestion in part-wise aggregations. A byproduct of our results is that the framework of low-congestion shortcuts interacts particularly well with the HYBRID model, as was also observed in [1].

1.2 Further Related Work

Our main reference point is the recent Laplacian solver of [18] with *existentially* almost-optimal complexity of $n^{o(1)}(\sqrt{n} + D)\log(1/\varepsilon)$ rounds, where $\varepsilon > 0$ represents the error of the solver. Specifically, they devised several new ideas and techniques to circumvent certain issues which mostly relate to the bandwidth restrictions of the CONGEST model; these building blocks, as well as the resulting Laplacian solver are revisited in our work to refine the performance of the solver. We are not aware of any previous research addressing this problem in the distributed context. On the other hand, the Laplacian paradigm has attracted a considerable amount of interest in the community of parallel algorithms [44, 6].

Research concerning hybrid communication networks in distributed algorithms was recently initiated by [3]. Specifically, they investigated the power of a model which integrates the standard LOCAL model [38] with the recently introduced node-capacitated clique (NCC) [2],

focusing mostly on distance computation tasks. Several of their results were subsequently improved and strengthened in subsequent works [35, 7] under the same model of computation. In our work we consider a substantially weaker model, imposing a severe limitation on the communication over the "local edges". This particular variant has been already studied in some recent works for a variety of fundamental problems [16, 24].

The NCC model, which captures the global network in all hybrid models studied thus far, was introduced in [2] partly to address the unrealistic power of the *congested clique* (CLIQUE) [39]. In the latter model each node can communicate *concurrently and independently* with *all* other nodes by $O(\log n)$-bit messages. In contrast, the NCC model allows communication with $O(\log n)$ (arbitrary) nodes per round. As a result, in the HYBRID model and under a sparse local network, only $\widetilde{\Theta}(n)$ bits can be exchanged overall per round, whereas CLIQUE allows for the exchange of up to $\widetilde{\Theta}(n^2)$ (distinct) bits. As evidence for the power of CLIQUE we note that even slightly super-constant lower bounds would give new lower bounds in circuit complexity, as implied by a simulation argument in [14]. Finally, we remark a subsequent work that leverages tools from the Laplacian paradigm in the *broadcast* variant of the congested clique [17].

2 Preliminaries

General notation. We denote with $[k] := \{1, 2, \dots, k\}$. Graphs throughout this paper are undirected. The nodes and the edges of a given graph G are denoted as $V(G)$ and $E(G)$, respectively. We also use $n := |V(G)|$ for brevity. The graphs are often weighted, in which case we assume (as is standard) that for all $e \in E(G), \boldsymbol{w}(e) \in \{1, 2, \dots, \text{poly}(n)\}$. We will denote the hop-diameter of a graph G with $D(G)$ (the hop-diameter ignores weights). Moreover, we use $A \uplus B$ to denote the multiset union, i.e., each element is repeated according to its multiplicity; this operation corresponds to disjoint unions when $A \cap B = \emptyset$.

Communication models. The communication network consists of a set of $\overline{n}$ entities with $[\overline{n}] := \{1, 2, \dots, \overline{n}\}$ being the set of their IDs, and a local communication *topology* given by a graph $\overline{G}$.[3] We define $D := D(\overline{G})$ to be the (hop-)diameter of the underlying network. At the beginning, each node knows its own unique $O(\log \overline{n})$-bit identifier as well as the weights of the incident edges. Communication occurs in *synchronous rounds*, and in every round nodes have unlimited computational power to process the information they possess. We will consider models with both *local* and *global* communication modes.

The *local* communication mode will be modeled with the *CONGEST model* [42] and *Supported-CONGEST model* [46], for which in each round every node can exchange an $O(\log \overline{n})$-bit message with each of its neighbors in $\overline{G}$ via the *local* edges. In the (standard) CONGEST model, each node $v \in V(\overline{G})$ initially only knows the identifiers of each node in v's own neighborhood, but has no further knowledge about the topology of the graph. On the other hand, in the Supported-CONGEST model, all nodes know the entire topology of $\overline{G}$ upfront, but not the input.

The *global* communication mode will be modeled using *NCC* [2], for which in each round every node can exchange $O(\log \overline{n})$-bit messages with $O(\log \overline{n})$ arbitrary nodes via *global* edges. If the capacity of some channel is exceeded, i.e., too many messages are sent to the

[3] To avoid any possible confusion we point out that, for consistency with the nomenclature of [18], we henceforth reserve $\overline{G}$ to denote the underlying *communication network*, while G is used in statements regarding arbitrary graphs.

same node, it will only receive an *arbitrary* (potentially adversarially selected) subset of the information based on the capacity of the network; the rest of the messages are dropped. In this context, we will let HYBRID be the integration of CONGEST and NCC (i.e., nodes have both a *local* and a *global* communication mode at their disposal).

The performance of a distributed algorithm will be measured in terms of its *round complexity* – the number of rounds required so that every node knows its part of the output. For randomized algorithms it will suffice to reach the desired state with high probability.[4] We will assume throughout this work that nodes have access to a common source of randomness; this comes without any essential loss of generality in our setting [19]. When talking about a distributed algorithm for a specific problem (e.g., Laplacian solving, part-wise aggregation, etc.) we assume the input is appropriately *distributedly stored* (i.e., each node will know its own part) and, upon termination, it will be required that the output is appropriately distributedly stored. The appropriate way to distributedly store the input and output will be explained in the problem definition.

Low-Congestion Shortcuts. A recurring scenario in distributed algorithms for global problems (e.g. MST) boils down to solving the following part-wise aggregation problem:

▶ **Definition 4** (Part-Wise Aggregation Problem). *Consider an n-node graph G whose node set $V(G)$ is* partitioned *into k (disjoint) parts $P_1 \uplus \cdots \uplus P_k \subseteq V(G)$ such that each induced subgraph $G[P_i]$ is* connected. *In the* part-wise aggregation *problem, each node $v \in V$ is given its part-ID (if any) and an $O(\log n)$-bit value $\boldsymbol{x}(v)$ as input. The goal is that, for every part P_i, all nodes in P_i learn the part-wise aggregate $\bigoplus_{w \in P_i} \boldsymbol{x}(w)$, where $\bigoplus$ is an arbitrary pre-defined* aggregation function.

Throughout this paper, we will assume that the aggregation function $\bigoplus$ is commutative and associative (e.g. min, sum, logical-AND), although this is not strictly needed (e.g., see [23]). To solve such problems, [20] introduced a natural combinatorial graph structure that they refer to as *low-congestion shortcuts.*

▶ **Definition 5** (Low-Congestion Shortcuts). *Consider a graph G whose node set $V(G)$ is* partitioned *into k (disjoint) parts $P_1 \uplus \cdots \uplus P_k \subseteq V(G)$ such that each induced subgraph $G[P_i]$ is* connected. *A collection of subgraphs $H_1, \ldots, H_k$ is a* shortcut *of G with* congestion *c and* dilation *d if the following properties hold: (i) the (hop) diameter of each subgraph $G[P_i] \cup H_i$ is at most d, and (ii) every edge is included in at most c many of the subgraphs H_i. The quantity $Q = c + d$ will be referred to as the* quality *of the shortcut.*

Importantly, a shortcut of quality Q allows us to solve the part-wise aggregation problem in $\widetilde{O}(Q)$ rounds of CONGEST, as formalized below.

▶ **Proposition 6.** *Suppose that $P_1, \ldots, P_k$ is any part-wise aggregation instance in a communication network $\overline{G}$. Given a shortcut of quality Q, we can solve with high probability the part-wise aggregation problem in $\widetilde{O}(Q)$ CONGEST rounds.*

Shortcut Quality and Construction of Shortcuts. Shortcut quality, introduced below, is a fundamental graph parameter that has been proven to characterize the complexity of many important problems in distributed computing.

[4] We say that an event holds with high probability if it occurs with probability at least $1 - 1/n^c$ for a (freely choosable) constant $c > 0$.

▶ **Definition 7.** *Given a graph $G = (V, E)$, we define the* shortcut quality $\mathrm{SQ}(G)$ *of G as the optimal (smallest) shortcut quality of the worst-case partition of V into disjoint and connected parts $P_1 \uplus P_2 \uplus \ldots \uplus P_k \subseteq V$.*

For fundamental problems such as MST, SSSP, and Min-Cut any correct algorithm requires $\widetilde{\Omega}(\mathrm{SQ}(\overline{G}))$ rounds on any network $\overline{G}$, even if we allow randomized solutions and (non-trivial) approximation factors. In fact, this limitation holds even when the network topology $\overline{G}$ is known to all nodes in advance [29]. We remark that $\widetilde{\Omega}(D(\overline{G})) \leq \mathrm{SQ}(\overline{G}) \leq O(D(\overline{G})+\sqrt{\overline{n}})$, and the upper bound is known to be tight in certain (pathological) worst-case graph instances.

Moreover, *assuming* fast distributed algorithms for constructing shortcuts of quality competitive with $\mathrm{SQ}(\overline{G})$, all of the aforementioned problems can be solved in $\widetilde{O}(\mathrm{SQ}(\overline{G}))$ rounds [20, 50, 23]. However, the key issue here is the algorithmic construction of the shortcuts upon which the above papers rely. While there has been a lot of recent progress in this regard, current algorithms are quite complicated and have sub-optimal guarantees. We recall below these state-of-the-art $\mathrm{SQ}(\overline{G})$-competitive construction results.

▶ **Theorem 8.** *There exists a distributed algorithm that, given any part-wise aggregation instance on any $\overline{n}$-node graph $\overline{G}$, computes with high probability a shortcut with the following guarantees:*

- *In CONGEST, the shortcut has quality* $\mathrm{poly}\big(\mathrm{SQ}(\overline{G})\big) \cdot \overline{n}^{o(1)}$ *and the algorithm terminates in* $\mathrm{poly}\big(\mathrm{SQ}(\overline{G})\big) \cdot \overline{n}^{o(1)}$ *rounds [27].*
- *In Supported-CONGEST, the shortcut has quality* $\widetilde{O}(\mathrm{SQ}(\overline{G}))$ *and the algorithm terminates in* $\widetilde{O}(\mathrm{SQ}(\overline{G}))$ *rounds [29].*

Universal Optimality. A distributed algorithm is said to be α-*universally optimal* if, on every network graph $\overline{G}$, it is α-competitive with the fastest correct algorithm on $\overline{G}$ [29]. Even the existence of such algorithms is not at all clear as it would seem possible that vastly different algorithms are required to leverage the structure of different networks. Nevertheless, a remarkable consequence of Theorem 8 is that in Supported-CONGEST we can design $\widetilde{O}(1)$-universally optimal algorithms for many fundamental optimization problems. Moreover, efficient shortcut construction is the only obstacle towards achieving these results in the full generality of CONGEST, which is an orthogonal issue and out of scope for this paper. Still, the aforementioned results are sufficient to design $\overline{n}^{o(1)}$-universally optimal algorithms on graphs that have shortcut quality $\mathrm{SQ}(\overline{G}) = \overline{n}^{o(1)}$.

Graphs Excluding Dense Minors. It turns out that the crucial issue of efficient shortcut construction can be resolved with a near-optimal, simple, and even deterministic algorithm for the rich class of graphs with *bounded minor density.* Formally, let us first recall the following definition.

▶ **Definition 9** (Minor Density). *The* minor density $\delta(G)$ *of a graph G is defined as*

$$\delta(G) = \max\left\{\frac{|E'|}{|V'|} : H = (V', E') \text{ is a minor of } G\right\}.$$

Any family of graphs closed under taking minors (such as planar graphs) has a constant minor density. For such graphs, [21] established efficient shortcut construction:

▶ **Theorem 10** ([21])**.** *Any graph G with hop-diameter D and minor density $\delta(G)$ admits shortcuts of quality $\widetilde{O}(\delta D)$, which can be constructed with high probability in $\widetilde{O}(\delta D)$ rounds of CONGEST.*

Some of our results apply for communication networks with *bounded treewidth*, so let us recall the following definition.

▶ **Definition 11** (Tree Decomposition and Treewidth). *A* tree decomposition *of a graph G is a tree T with tree-nodes $X_1, \dots, X_k$, where each X_i is a subset of $V(G)$ satisfying the following properties:*
1. $V = \bigcup_{i=1}^{k} X_i$*;*
2. *For any node $u \in V(G)$, the tree-nodes containing u form a connected subtree of T;*
3. *For every edge $\{u, v\} \in E(G)$, there exists a tree-node X_i which contains both u and v.*

The width *w of the tree decomposition is defined as $w := \max_{i\in[k]} |X_i| - 1$. Moreover, the* treewidth *$\mathrm{tw}(G)$ of G is defined as the minimum of the width among all possible tree decompositions of G.*

Bounded-treewidth graphs inherit all of the nice properties guaranteed by Theorem 10, as implied by the following well-known fact.

▶ **Lemma 12.** *For any graph G, $\delta(G) \leq \mathrm{tw}(G)$.*

3 The Congested Part-Wise Aggregation Problem

This section is concerned with a *congested* generalization of the standard part-wise aggregation problem (Definition 4), formally introduced below.

▶ **Definition 13** (Congested Part-Wise Aggregation Problem). *Consider an n-node graph G with a collection of k subsets of nodes $P_1, \dots, P_k \subseteq V(G)$ called* parts *such that each induced subgraph $G[P_i]$ is* connected *and each node $v \in V(G)$ is contained in at most $\rho \in \mathbb{Z}_{\geq 1}$ many parts, i.e., $\forall v \in V(G) \quad |\{i : P_i \ni v\}| \leq \rho$. In the ρ-*congested part-wise aggregation *problem, each node v is given the following as input: for each part $P_i \ni v$ node v knows the part-ID i and an $O(\log n)$-bit part-specific value $\boldsymbol{x}_i(v)$. The goal is that, for each part P_i, all nodes in P_i learn the part-wise aggregate $\bigoplus_{w\in P_i} \boldsymbol{x}_i(w)$, where $\bigoplus$ is a pre-defined* aggregation function.

This congested generalization of the standard part-wise aggregation problem that we study in this section turns out to be a central ingredient in our refined Laplacian solver; this is further explained in Section 4. The remainder of this section is organized as follows. In Section 3.1 we establish near-optimal algorithms for solving congested part-wise aggregations in CONGEST, which is also the main focus of this section. We conclude by pointing out the construction for NCC in Section 3.2. Due to space limitations, all the omitted proofs are deferred to the full version of this paper.

3.1 Solving Congested Instances in the CONGEST Model

The first natural strategy for solving the ρ-congested part-wise aggregation problem of Definition 13 is through a reduction to $\mathrm{poly}(\rho)$ 1-congested instances. However, this approach immediately fails even if we allow $\rho = 2$. Indeed, there exist congested part-wise aggregation instances for which every two (distinct) parts share a common node, even when $\rho = 2$, leading to the following observation.

▶ **Observation 14.** *For an infinite family of values $\overline{n}$, there exists an $\overline{n}$-node planar graph $\overline{G}$ and a 2-congested part-wise aggregation instance $\mathcal{I}$ with $k = \Theta(\sqrt{\overline{n}})$ parts such that reducing $\mathcal{I}$ to the union of k' 1-congested part-wise aggregation instances on $\overline{G}$ requires $k' = \Omega(\sqrt{\overline{n}})$.*

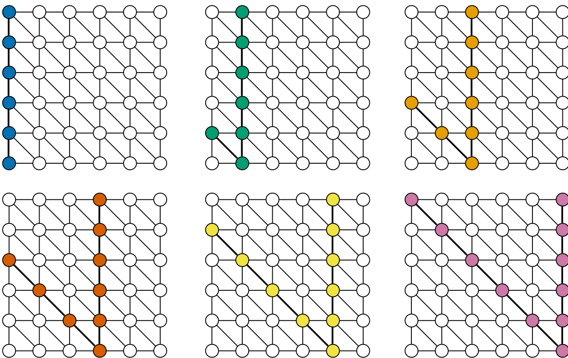


Figure 1 A 2-congested part-wise aggregation problem on a 6×6 grid (the instance immediately extends to an $\sqrt{\overline{n}} \times \sqrt{\overline{n}}$ topology). Different colors highlight different parts of the instance.

Such a pattern is illustrated in Figure 1. Indeed, in that 2-congested part-wise aggregation instance every two distinct parts share a common node. As a result, directly employing a 1-congested part-wise aggregation oracle is of little use since it would introduce an overhead depending on the number of parts. In light of this, we develop a more refined approach that leverages what we refer to as the *layered graph*.

3.1.1 The Layered Graph

Here we introduce the *layered graph* $\widehat{G}_\rho$, associated with the underlying graph $\overline{G}$. Then, we reduce any ρ-congested part-wise aggregation on $\overline{G}$ to a 1-congested instance on $\widehat{G}_{O(\rho)}$.

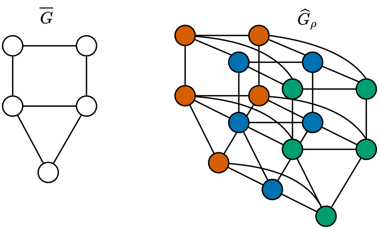


Figure 2 An example of a transformation from $\overline{G}$ to the layered graph $\widehat{G}_\rho$ with $\rho = 3$. We have highlighted with different colors different layers of the graph.

The Layered Graph. Consider an underlying network $\overline{G}$ and some $\rho \in \mathbb{Z}_{\geq 1}$, corresponding to the congestion parameter in Definition 13. The *layered graph* $\widehat{G}_\rho$ is constructed in the following way. First, we let $\widehat{G}_\rho$ be a disjoint union of ρ copies of $\overline{G}$ (called *layers*), namely $\overline{G}_1, \overline{G}_2, \dots, \overline{G}_\rho$. Each node $v \in V(\overline{G})$ is associated with its copies $v_1, v_2, \dots, v_\rho \in V(\widehat{G}_\rho)$. We also add an edge between each two copies that originate from the same node (i.e., we add a clique to $\widehat{G}_\rho$ on the set of copies associated with the same node $v \in V(\overline{G})$); this construction is illustrated in Figure 2. The layered graph induces a natural *projection* operation $\pi : V(\widehat{G}_\rho) \to V(\overline{G})$ which maps a copy v_i to its original node $v = \pi(v_i)$. Furthermore, we often talk about simulating $\widehat{G}_\rho$ in $\overline{G}$, by which we mean that each node v simulates – learns all the inputs and can generate all outputs – for its copies $v_1, \dots, v_\rho$. Throughout this paper, we will assume that $\rho = \text{poly}(\overline{n})$ so that any $O(\log n)$-bit message on $\widehat{G}_\rho$ can be sent within $O(1)$ rounds in $\overline{G}$; this also keeps the $\widetilde{O}$-notation well-defined.

The main goal of this section is to establish that the ρ-congested part-wise aggregation problem on $\overline{G}$ can be reduced to a 1-congested instance on $\widehat{G}_{O(\rho)}$, as formalized below.

▶ **Lemma 15** (Unrestricted Congested Part-Wise Aggregation). *Let $\overline{G}$ be an $\overline{n}$-node graph and let $\mathbb{Z}_{\geq 1} \ni \rho \leq \mathrm{poly}(\overline{n})$. Suppose that any (1-congested) part-wise aggregation on $\widehat{G}_{O(\rho)}$ can be solved with a τ-round CONGEST algorithm on $\widehat{G}_{O(\rho)}$. Then, there exists an $\widetilde{O}(\rho \cdot \tau)$-round CONGEST algorithm on $\overline{G}$ that solves any ρ-congested part-wise aggregation instance on $\overline{G}$.*

Towards establishing this reduction, we first point out that any CONGEST algorithm on $\widehat{G}_\rho$ can be simulated with only a ρ multiplicative overhead in the round complexity.

▶ **Lemma 16** (Simulating $\widehat{G}_\rho$ in $\overline{G}$). *For any $\overline{G}$ and any $\mathbb{Z}_{\geq 1} \ni \rho \leq \mathrm{poly}(\overline{n})$, we can simulate any τ-round CONGEST algorithm on $\widehat{G}_\rho$ with a $(\rho \cdot \tau)$-round CONGEST algorithm on $\overline{G}$.*

Furthermore, we will use a folklore result showing how to color a (multi)graph of maximum degree Δ in $O(\Delta)$ colors in $O(\log n)$ rounds of CONGEST. By multigraph here we simply mean that there can be multiple parallel edges between the same pair of nodes, and every such edge can carry an independent message per round.

▶ **Lemma 17** (Folklore, [30]). *Given a (multi)graph G with n nodes and maximum degree $\Delta \leq \mathrm{poly}(n)$, there exists a randomized CONGEST algorithm that colors the edges of G with $O(\Delta)$ colors and completes in $O(\log n)$ rounds, with high probability. The coloring is proper, i.e., two edges that share an endpoint are assigned a different color.*

Using this lemma, we first prove a version of our main reduction (Lemma 15), but with the slight twist that we restrict each part of the ρ-congested part-wise aggregation problem to be a simple path.

▶ **Lemma 18** (Path-Restricted Congested Part-Wise Aggregation). *Let $\overline{G}$ be an $\overline{n}$-node graph and let $\mathbb{Z}_{\geq 1} \ni \rho \leq \mathrm{poly}(\overline{n})$. Suppose that there exists a τ-round CONGEST algorithm solving the (1-congested) part-wise aggregation on $\widehat{G}_{O(\rho)}$. Then, there exists an $\widetilde{O}(\rho \cdot \tau)$-round CONGEST algorithm on $\overline{G}$ that solves any ρ-congested part-wise aggregation instance on $\overline{G}$ when each part is* restricted to be a simple path[5] *(nodes are not repeated in simple paths).*

Finally, our reduction in Lemma 15 follows by reformulating [29, Lemma 7.2].

3.1.2 Treewidth-Bounded Graphs

Here we leverage the reduction we established in Lemma 15 to obtain a simple algorithm for solving the congested part-wise aggregation problem in treewidth-bounded graphs. The crucial observation is that the treewidth of the layered graph can only grow by a factor of ρ compared to the treewidth of the underlying graph, as we show below.

▶ **Lemma 19.** *If the treewidth of $\overline{G}$ is $\mathrm{tw}(\overline{G})$, then $\mathrm{tw}(\widehat{G}_\rho) \leq \rho\, \mathrm{tw}(\overline{G}) + \rho - 1$.*

Combining this guarantee with Lemmas 12 and 15 and Theorem 10, we obtain the following immediate consequence.

▶ **Corollary 20.** *Let $\overline{G}$ be an $\overline{n}$-node communication network of diameter at most D and treewidth $\mathrm{tw}(\overline{G})$. Then, we can solve with high probability any ρ-congested part-wise aggregation problem in $\overline{G}$ within $\widetilde{O}(\rho^2 \cdot \mathrm{tw}(\overline{G}) \cdot D)$ rounds of CONGEST.*

---

[5] I.e., there exists a simple path traversing all the nodes of the part, and each node knows the corresponding incident edges of that path.

Minor Density in the Layered Graph. In light of Lemma 19, a natural question is whether an analogous bound holds with respect to the minor density of the underlying graph; i.e., whether $\delta(\widehat{G}_\rho) = \text{poly}(\rho)\delta(G)$. Unfortunately, this is not possible, as illustrated in Figure 3.

▶ **Observation 21.** *There exists an n-node graph G with minor density $\delta(G) = \widetilde{O}(1)$, but its 2-layered version $\widehat{G}_2$ has minor density $\delta(\widehat{G}_2) = \Omega(\sqrt{n})$.*

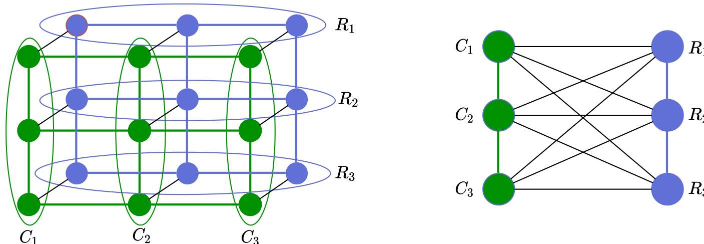


Figure 3 The layered graph $\widehat{G}_\rho$ of a 3×3 grid with every node having congestion $\rho = 2$ (left), and a minor of $\widehat{G}_\rho$ induced by the connected components $\{C_1, C_2, C_3, R_1, R_2, R_3\}$ (right).

3.1.3 General Graphs

We conclude with our main result of Section 3.1: a near-optimal distributed algorithm for solving the ρ-congested part-wise aggregation problem in general graphs. In light of our reduction in Lemma 15, the technical crux is to control the degradation in the shortcut quality incurred by the transformation into the layered graph. Surprisingly, we show that the shortcut quality of $\widehat{G}_\rho$ does not increase by more than a polylogarithmic factor even when the number of layers is polynomial:

▶ **Theorem 22.** *For any $\overline{n}$-node graph $\overline{G}$ and any $\mathbb{Z}_{\geq 1} \ni \rho \leq \text{poly}(\overline{n})$, we have that $\text{SQ}(\widehat{G}_\rho) = \widetilde{O}(\text{SQ}(\overline{G}))$.*

This theorem improves over our previous result for treewidth-bounded graphs (Lemma 19) since the latter guarantee inevitably induces a linear factor of ρ in the shortcut quality of $\widehat{G}_\rho$. While this will not affect the asymptotic performance of the Laplacian solver, this improvement might prove to be important for future applications. Assuming that we have shown Theorem 22, we can then utilize the efficient shortcut constructions given in Theorem 8 to solve ρ-congested part-wise aggregations on any graph.

▶ **Corollary 23.** *There exists a randomized distributed algorithm that, for any $\overline{n}$-node graph $\overline{G}$ and $\rho \in \mathbb{Z}_{\geq 1} \leq \text{poly}(\overline{n})$, solves with high probability any ρ-congested part-wise aggregation instance on $\overline{G}$ with the following guarantees:*

- *In the CONGEST, the algorithm terminates in at most $\rho \cdot \text{poly}\big(\text{SQ}(\overline{G})\big) \cdot \overline{n}^{o(1)}$ rounds.*
- *In the CONGEST model on graphs with minor density δ, it requires $\widetilde{O}(\rho \cdot \delta \cdot D)$ rounds.*
- *In the Supported-CONGEST, the algorithm terminates in $\widetilde{O}(\rho \cdot \text{SQ}(\overline{G}))$ rounds.*

The rest of this subsection is dedicated to the proof of Theorem 22. First, to argue about the shortcut quality of the layered graph, we need to develop several generalized notions of node connectivity.

Pair Node Connectivity. Given a (multi)set of source-sink pairs $\mathcal{P} = \{(s_i, t_i)\}_{i=1}^k$ in G, we say that $\mathcal{P}$ has pair node connectivity ρ if there exist paths $P_1, \ldots, P_k$, with s_i and t_i being the endpoints of each P_i, such that every node $v \in V(G)$ is contained in at most ρ many paths, i.e., for all v we have $|\{i : V(P_i) \ni v\}| \leq \rho$. If $\mathcal{P}$ has pair node connectivity 1 we say that they are pair *node-disjointly connectable*.

Any-to-Any Node Connectivity. Suppose that we are given multisets of k *sources* $S = \{s_1, \ldots, s_k\}$ and k *sinks* $T = \{t_1 \ldots, t_k\}$. We say that (S, T) have any-to-any node connectivity ρ if there is a permutation $\pi : \{1, \ldots, k\} \to \{1, \ldots, k\}$ such that the pairs $\{(s_i, t_{\pi(i)})\}_{i=1}^k$ have pair node connectivity ρ. If (S, T) have any-to-any node connectivity 1 we say they are any-to-any *node-disjointly connectable*.

The following decomposition lemma states that two sets with any-to-any node connectivity ρ can be decomposed into $\widetilde{O}(\rho)$ many pairs of subsets that are any-to-any node-disjointly connectable.

▶ **Lemma 24.** *Given a graph G, suppose we are given any two multisets of nodes $S \subseteq V(G)$ and $T \subseteq V(G)$ of size $k := |S| = |T|$ that have any-to-any node connectivity ρ. Then, we can partition $S = S_1 \uplus S_2 \uplus \ldots \uplus S_{O(\rho \log k)}$ and $T = T_1 \uplus T_2 \uplus \ldots T_{O(\rho \log k)}$ such that $|S_i| = |T_i|$ and (S_i, T_i) are any-to-any node-disjointly connectable.*

Next, we introduce two communication tasks that will be useful for characterizing the shortcut quality.

Multiple-Unicast Problem. Suppose that we are given k source-sink pairs $\mathcal{P} = \{(s_i, t_i)\}_{i=1}^k$. The goal is to find the smallest possible *completion time* τ such that there are k paths $P_1, \ldots, P_k$ for which (1) the endpoints of each P_i are exactly s_i and t_i; (2) the dilation is τ, i.e., each path P_i has at most τ hops; and (3) the congestion is τ, i.e., each edge $e \in E(G)$ is contained in at most τ many paths.

Any-to-Any-Cast Problem. Suppose we are given k *sources* $S = \{s_1, \ldots, s_k\}$ and k *sinks* $T = \{t_1 \ldots, t_k\}$. The goal is to find the smallest *completion time* τ such that there exists a permutation $\pi : \{1, \ldots, k\} \to \{1, \ldots, k\}$ for which the multiple-unicast problem on $\{(s_i, t_{\pi(i)})\}_{i=1}^k$ has a completion time of at most τ.

Finally, we now recall (a reinterpretation of) a result characterizing shortcut quality from [28, 29]. Shortcut quality was originally defined as the smallest completion-time of the worst-case generalized (with respect to parts) multiple-unicast (i.e., multi-commodity) problem over a *pair* node-disjointly connectable instance (Definition 7). Using recent network coding gap results, we can equivalently express shortcut quality as the smallest completion-time of the worst-case any-to-any-cast (i.e., single-commodity) problem over sources and sinks that are *any-to-any* node-disjointly connectable. The formal statement follows.

▶ **Theorem 25** ([28, 29]). *Consider any graph G and let τ be the worst-case completion time of any-to-any-cast problems taken over all any-to-any node-disjointly connectable sets $(S \subseteq V(G), T \subseteq V(G))$. Then, $\tau = \widetilde{\Theta}(\mathrm{SQ}(G))$.*

Finally, combining all of the previous ingredients, we are ready to show Theorem 22.

Proof of Theorem 22. Let $S \subseteq V(\widehat{G}_\rho)$ and $T \subseteq V(\widehat{G}_\rho)$ be any-to-any node-disjointly connectable sets such that the completion time of any-to-any-cast between S and T is $\widetilde{\Theta}(\mathrm{SQ}(\widehat{G}_\rho))$ (Theorem 25). Let $k := |S| = |T|$, and suppose that $S' := \biguplus_{s \in S}\{\pi(s)\} \subseteq V(\overline{G})$

and $T' := \biguplus_{t \in T}\{\pi(t)\} \subseteq V(\overline{G})$ are the multisets induced by projecting S and T to $\overline{G}$, respectively. By construction of $\widehat{G}_\rho$, S' and T' have any-to-any node connectivity ρ; to see this, consider the witness paths disjointly connecting them in $\widehat{G}_\rho$ and project them to $\overline{G}$. Therefore, we can partition $S' = S'_1 \uplus \ldots \uplus S'_{O(\rho \log k)}$ and $T' = T'_1 \uplus \ldots \uplus T'_{O(\log k)}$ such that $|S'_i| = |T'_i|$ and (S'_i, T'_i) are any-to-any node-disjointly connectable in $\overline{G}$ (Lemma 24).

By definition of shortcut quality, for each $i \in \{1, \ldots, O(\rho \log k)\}$ there exists a set of paths $(P^i_j)_{j=1}^{|S'_i|}$ in $\overline{G}$ between S'_i and T'_i of quality (i.e., both congestion and dilation) at most $\mathrm{SQ}(\overline{G})$. Then, we inject the first $O(\log k)$ collections of paths $(P^1_j)_j, (P^2_j)_j, \ldots, (P^{O(\log k)}_j)_j$ to the first layer $\overline{G}_1$ of $\widehat{G}_\rho$; the second $O(\log k)$ collections to the second layer $\overline{G}^2$, and so on, until we finally inject the last $O(\log k)$ collections to the last layer $\overline{G}_\rho$. Note that only the paths on the same layer interact, so both the congestion and dilation after injecting all paths into $\widehat{G}_\rho$ is $O(\mathrm{SQ}(\overline{G}) \log k)$. Hence, the same applies for the shortcut quality. Finally, to solve the any-to-any-cast problem on S and T one might need to add an between-layer edge at the beginning and at the end since each injected path is restricted to some adversarially chosen layer. However, this only increases the congestion and dilation by $O(1)$. Hence, the completion time of any-to-any-cast between S and T is $\widetilde{O}(\mathrm{SQ}(\overline{G}))$, implying that $\mathrm{SQ}(\widehat{G}_\rho) = \widetilde{O}(\mathrm{SQ}(\overline{G}))$. ◀

3.2 The NCC Model

We next turn our attention to the NCC model. We observe that the ρ-congested part-wise aggregation problem admits a solution in $\mathrm{poly}(\rho, \log \overline{n})$ rounds of NCC. This is established after appropriately translating the communication primitives established for NCC in [2].

▶ **Lemma 26.** *Let $\overline{G}$ be an $\overline{n}$-node graph. Then, we can solve with high probability any ρ-congested part-wise aggregation problem on $\overline{G}$ after $O(\rho + \log \overline{n})$ rounds of NCC.*

4 Almost Universally Optimal Laplacian Solvers

In this section, we relate the congested part-wise aggregation problem we studied in the previous section with the Laplacian solver in [18]. To present a unifying analysis for both CONGEST and HYBRID, as well as for future applications and extensions, we analyze the distributed Laplacian solver under the following hypothesis.

▶ **Assumption 27.** *Consider a model of computation which incorporates CONGEST. We assume that we can solve with high probability any ρ-congested part-wise aggregation problem in $Q(\rho) = O(\rho^c Q(1))$ rounds, for some universal constant $c \geq 1$.*

One of our crucial observations is that the performance of the Laplacian solver of [18] can be parameterized in terms of the complexity of the congested part-wise aggregation problem. Indeed, we revisit and refine the main building blocks of their solver, leading to the following.

▶ **Theorem 28.** *Consider a weighted $\overline{n}$-node graph $\overline{G}$ for which Assumption 27 holds for some $Q(\rho) = O(\rho^c \mathcal{Q})$, where c is a universal constant and $\mathcal{Q} = \mathcal{Q}(\overline{G})$ is some parameter. Then, we can solve any Laplacian system after $\overline{n}^{o(1)} \mathcal{Q} \log(1/\varepsilon)$ rounds.*

Combining this theorem with Corollary 23 and Lemma 26 yields the following immediate consequences.

▶ **Theorem 2.** *Consider any n-node graph G with shortcut quality $\mathrm{SQ}(G)$ and hop-diameter D. There exists a distributed Laplacian solver with error $\varepsilon > 0$ with the following guarantees:*

- *In the Supported-CONGEST model, it requires $n^{o(1)}\,\mathrm{SQ}(G)\log(1/\varepsilon)$ rounds.*
- *In the CONGEST model, it requires $n^{o(1)}\,\mathrm{poly}(\mathrm{SQ}(G))\log(1/\varepsilon)$ rounds.*
- *In the CONGEST model on graphs with minor density δ, it requires $n^{o(1)}\delta D\log(1/\varepsilon)$ rounds.*

▶ **Theorem 3.** *Consider any n-node graph. There exists a distributed Laplacian solver in the HYBRID model with round complexity $n^{o(1)}\log(1/\varepsilon)$, where $\varepsilon > 0$ is the error of the solver.*

Lower Bound in Supported-CONGEST. Finally, we complement our positive results with an almost-matching lower bound on any graph $\overline{G}$, applicable even under the Supported-CONGEST model, thereby establishing universal optimality up to an $\overline{n}^{o(1)}$ factor. Our reduction leverages the refined hardness result established in [29] for the *spanning connected subgraph* problem [13]. In this problem a subgraph $\overline{H}$ of $\overline{G}$ is specified with nodes knowing all of the incident edges belonging to $\overline{H}$. The goal is to let every node learn whether $\overline{H}$ is connected and spans the entire network.

▶ **Theorem 29** ([29]). *Let $\mathcal{A}$ be any algorithm which is always correct with probability*[6] *at least $\frac{2}{3}$ for the spanning connected subgraph problem, and $T(\overline{G}) = \max_{\mathcal{I}} T_{\mathcal{A}}(\mathcal{I};\overline{G})$ be the worst-case round-complexity of $\mathcal{A}$ under $\overline{G}$. Then, $T(\overline{G}) = \widetilde{\Omega}(\mathrm{SQ}(\overline{G}))$.*

In this context, we show that a Laplacian solver can be leveraged to solve the spanning connected subgraph problem, leading to the following lower bound.

▶ **Theorem 1.** *Consider a graph $\overline{G}$ with shortcut quality $\mathrm{SQ}(\overline{G})$. Then, solving a Laplacian system on $\overline{G}$ with $\varepsilon \le \frac{1}{2}$ requires $\widetilde{\Omega}(\mathrm{SQ}(\overline{G}))$ rounds in both CONGEST and Supported-CONGEST models.*

5 Conclusions

In this paper, we have established almost universally optimal Laplacian solvers for both the (Supported-)CONGEST and the HYBRID model. One of our main technical contributions was to introduce and study a congested generalization of the standard part-wise aggregation problem, which we believe may find further applications beyond the Laplacian paradigm in the future. For example, one candidate problem would be to refine the distributed algorithm for max-flow due to [22].

We also hope that our accelerated Laplacian solvers will be used as a basic primitive for obtaining improved distributed algorithms for other fundamental optimization problems as well. For example, our results directly imply an exact $O(m^{1/2+o(1)}\,\mathrm{SQ}(G))$ algorithm for the max-flow problem using a standard reduction [12]. On the other hand, there are substantial obstacles in obtaining further improvements and strengthening the max-flow algorithm of [18], which in turn relies on the more recent techniques of [40, 10], as that would require solving exactly the single-source shortest paths problem in almost $\mathrm{SQ}(G)$ rounds.

[6] Note that [29] only proved this for always-correct algorithms with probability 1, but the extension we claim here follows readily from their argument.

References

1. Ioannis Anagnostides and Themis Gouleakis. Deterministic distributed algorithms and lower bounds in the hybrid model. In *35th International Symposium on Distributed Computing, DISC 2021*, volume 209 of *LIPIcs*, pages 5:1–5:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.DISC.2021.5.
2. John Augustine, Mohsen Ghaffari, Robert Gmyr, Kristian Hinnenthal, Christian Scheideler, Fabian Kuhn, and Jason Li. Distributed computation in node-capacitated networks. In Christian Scheideler and Petra Berenbrink, editors, *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019*, pages 69–79. ACM, 2019. doi:10.1145/3323165.3323195.
3. John Augustine, Kristian Hinnenthal, Fabian Kuhn, Christian Scheideler, and Philipp Schneider. Shortest paths in a hybrid network model. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020*, pages 1280–1299. SIAM, 2020. doi:10.1137/1.9781611975994.78.
4. Kyriakos Axiotis, Aleksander Madry, and Adrian Vladu. Circulation control for faster minimum cost flow in unit-capacity graphs. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*, pages 93–104. IEEE, 2020. doi:10.1109/FOCS46700.2020.00018.
5. Kyriakos Axiotis, Aleksander Madry, and Adrian Vladu. Faster sparse minimum cost flow by electrical flow localization. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021*, pages 528–539. IEEE, 2021. doi:10.1109/FOCS52979.2021.00059.
6. Guy E. Blelloch, Anupam Gupta, Ioannis Koutis, Gary L. Miller, Richard Peng, and Kanat Tangwongsan. Nearly-linear work parallel SDD solvers, low-diameter decomposition, and low-stretch subgraphs. *Theory Comput. Syst.*, 55(3):521–554, 2014. doi:10.1007/s00224-013-9444-5.
7. Keren Censor-Hillel, Dean Leitersdorf, and Volodymyr Polosukhin. Distance computations in the hybrid network model via oracle simulations. In *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021*, volume 187 of *LIPIcs*, pages 21:1–21:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.STACS.2021.21.
8. Keren Censor-Hillel, Dean Leitersdorf, and Volodymyr Polosukhin. On sparsity awareness in distributed computations. In Kunal Agrawal and Yossi Azar, editors, *SPAA '21: 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 151–161. ACM, 2021. doi:10.1145/3409964.3461798.
9. Tao Chen, Xiaofeng Gao, and Guihai Chen. The features, hardware, and architectures of data center networks: A survey. *Journal of Parallel and Distributed Computing*, 96:45–74, 2016. doi:10.1016/j.jpdc.2016.05.009.
10. Michael B. Cohen, Aleksander Madry, Piotr Sankowski, and Adrian Vladu. Negative-weight shortest paths and unit capacity minimum cost flow in õ $(m^{10/7} \log W)$ time (extended abstract). In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017*, pages 752–771. SIAM, 2017. doi:10.1137/1.9781611974782.48.
11. Sam Coy, Artur Czumaj, Michael Feldmann, Kristian Hinnenthal, Fabian Kuhn, Christian Scheideler, Philipp Schneider, and Martijn Struijs. Near-shortest path routing in hybrid communication networks, 2022. arXiv:2202.08008.
12. Samuel I. Daitch and Daniel A. Spielman. Faster approximate lossy generalized flow via interior point algorithms. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, 2008*, pages 451–460. ACM, 2008. doi:10.1145/1374376.1374441.
13. Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, STOC '11, pages 363–372, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1993636.1993686.

14 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In Magnús M. Halldórsson and Shlomi Dolev, editors, *ACM Symposium on Principles of Distributed Computing, PODC '14*, pages 367–376. ACM, 2014.

15 Michael Elkin. Unconditional lower bounds on the time-approximation tradeoffs for the distributed minimum spanning tree problem. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '04, pages 331–340, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/1007352.1007407.

16 Michael Feldmann, Kristian Hinnenthal, and Christian Scheideler. Fast hybrid network algorithms for shortest paths in sparse graphs. In *24th International Conference on Principles of Distributed Systems, OPODIS 2020*, volume 184 of *LIPIcs*, pages 31:1–31:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.OPODIS.2020.31.

17 Sebastian Forster and Tijn de Vos. The laplacian paradigm in the broadcast congested clique. In *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, 2022*, pages 335–344. ACM, 2022. doi:10.1145/3519270.3538436.

18 Sebastian Forster, Gramoz Goranci, Yang P. Liu, Richard Peng, Xiaorui Sun, and Mingquan Ye. Minor sparsifiers and the distributed laplacian paradigm. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021*, pages 989–999. IEEE, 2021. doi:10.1109/FOCS52979.2021.00099.

19 Mohsen Ghaffari. *Near-Optimal Scheduling of Distributed Algorithms*, pages 3–12. Association for Computing Machinery, New York, NY, USA, 2015. doi:10.1145/2767386.2767417.

20 Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks II: low-congestion shortcuts, mst, and min-cut. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016*, pages 202–219. SIAM, 2016. doi:10.1137/1.9781611974331.ch16.

21 Mohsen Ghaffari and Bernhard Haeupler. Low-congestion shortcuts for graphs excluding dense minors. In *PODC '21: ACM Symposium on Principles of Distributed Computing, 2021*, pages 213–221. ACM, 2021. doi:10.1145/3465084.3467935.

22 Mohsen Ghaffari, Andreas Karrenbauer, Fabian Kuhn, Christoph Lenzen, and Boaz Patt-Shamir. Near-optimal distributed maximum flow: Extended abstract. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015*, pages 81–90. ACM, 2015. doi:10.1145/2767386.2767440.

23 Mohsen Ghaffari and Goran Zuzic. Universally-optimal distributed exact min-cut. In *PODC '22: ACM Symposium on Principles of Distributed Computing, 2022*, pages 281–291. ACM, 2022. doi:10.1145/3519270.3538429.

24 Thorsten Götte, Kristian Hinnenthal, Christian Scheideler, and Julian Werthmann. Time-optimal construction of overlay networks. In *PODC '21: ACM Symposium on Principles of Distributed Computing*, pages 457–468. ACM, 2021. doi:10.1145/3465084.3467932.

25 Bernhard Haeupler, Taisuke Izumi, and Goran Zuzic. Low-congestion shortcuts without embedding. In George Giakkoupis, editor, *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016*, pages 451–460. ACM, 2016. doi:10.1145/2933057.2933112.

26 Bernhard Haeupler, Taisuke Izumi, and Goran Zuzic. Near-optimal low-congestion shortcuts on bounded parameter graphs. In *Distributed Computing – 30th International Symposium, DISC 2016*, volume 9888 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2016. doi:10.1007/978-3-662-53426-7_12.

27 Bernhard Haeupler, Harald Räcke, and Mohsen Ghaffari. Hop-constrained expander decompositions, oblivious routing, and distributed universal optimality. In *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, 2022*, pages 1325–1338. ACM, 2022. doi:10.1145/3519935.3520026.

28 Bernhard Haeupler, David Wajc, and Goran Zuzic. Network coding gaps for completion times of multiple unicasts. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 494–505. IEEE, 2020. doi:10.1109/FOCS46700.2020.00053.

29 Bernhard Haeupler, David Wajc, and Goran Zuzic. Universally-optimal distributed algorithms for known topologies. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 1166–1179. ACM, 2021. doi:10.1145/3406325.3451081.

30 Öjvind Johansson. Simple distributed $\delta+1$-coloring of graphs. *Information Processing Letters*, 70(5):229–232, 1999.

31 Udit Narayana Kar and Debarshi Kumar Sanyal. An overview of device-to-device communication in cellular networks. *ICT Express*, 4(4):203–208, 2018. doi:10.1016/j.icte.2017.08.002.

32 Jonathan A. Kelner, Yin Tat Lee, Lorenzo Orecchia, and Aaron Sidford. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. In Chandra Chekuri, editor, *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014*, pages 217–226. SIAM, 2014. doi:10.1137/1.9781611973402.16.

33 Jonathan A. Kelner, Lorenzo Orecchia, Aaron Sidford, and Zeyuan Allen Zhu. A simple, combinatorial algorithm for solving SDD systems in nearly-linear time. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *Symposium on Theory of Computing Conference, STOC'13, 2013*, pages 911–920. ACM, 2013. doi:10.1145/2488608.2488724.

34 Ioannis Koutis, Gary L. Miller, and Richard Peng. Approaching optimality for solving SDD linear systems. *SIAM J. Comput.*, 43(1):337–354, 2014. doi:10.1137/110845914.

35 Fabian Kuhn and Philipp Schneider. Computing shortest paths and diameter in the hybrid network model. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, pages 109–118. Association for Computing Machinery, 2020.

36 Fabian Kuhn and Philipp Schneider. Routing schemes and distance oracles in the hybrid model, 2022. arXiv:2202.06624.

37 Rasmus Kyng and Sushant Sachdeva. Approximate gaussian elimination for laplacians – fast, sparse, and simple. In Irit Dinur, editor, *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016*, pages 573–582. IEEE Computer Society, 2016. doi:10.1109/FOCS.2016.68.

38 Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992.

39 Zvi Lotker, Elan Pavlov, Boaz Patt-Shamir, and David Peleg. MST construction in $O(\log\log n)$ communication rounds. In Arnold L. Rosenberg and Friedhelm Meyer auf der Heide, editors, *SPAA 2003: Proceedings of the Fifteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, 2003*, pages 94–100. ACM, 2003. doi:10.1145/777412.777428.

40 Aleksander Madry. Computing maximum flow with augmenting electrical flows. In Irit Dinur, editor, *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016*, pages 593–602. IEEE Computer Society, 2016. doi:10.1109/FOCS.2016.70.

41 D. Peleg and V. Rubinovich. A near-tight lower bound on the time complexity of distributed mst construction. In *40th Annual Symposium on Foundations of Computer Science*, pages 253–261, 1999. doi:10.1109/SFFCS.1999.814597.

42 David Peleg. *Distributed computing: a locality-sensitive approach.* SIAM, 2000.

43 Richard Peng. Approximate undirected maximum flows in $O(m\text{polylog}(n))$ time. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016*, pages 1862–1867. SIAM, 2016. doi:10.1137/1.9781611974331.ch130.

44 Richard Peng and Daniel A. Spielman. An efficient parallel solver for SDD linear systems. In David B. Shmoys, editor, *Symposium on Theory of Computing, STOC 2014*, pages 333–342. ACM, 2014.

45 Václav Rozhon, Christoph Grunau, Bernhard Haeupler, Goran Zuzic, and Jason Li. Undirected $(1+\epsilon)$-shortest paths via minor-aggregates: near-optimal deterministic parallel and distributed algorithms. In *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, 2022*, pages 478–487. ACM, 2022. doi:10.1145/3519935.3520074.

46 Stefan Schmid and Jukka Suomela. Exploiting locality in distributed sdn control. *HotSDN 2013 – Proceedings of the 2013 ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, pages 121–126, 2013. doi:10.1145/2491185.2491198.

47 Daniel A. Spielman and Shang-Hua Teng. Nearly linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems. *SIAM J. Matrix Anal. Appl.*, 35(3):835–885, 2014. doi:10.1137/090771430.

48 Jan van den Brand, Yin Tat Lee, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Bipartite matching in nearly-linear time on moderately dense graphs. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*, pages 919–930. IEEE, 2020. doi:10.1109/FOCS46700.2020.00090.

49 Guohui Wang, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, T.S. Eugene Ng, Michael Kozuch, and Michael Ryan. C-through: Part-time optics in data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 327–338. Association for Computing Machinery, 2010. doi:10.1145/1851182.1851222.

50 Goran Zuzic, Gramoz Goranci, Mingquan Ye, Bernhard Haeupler, and Xiaorui Sun. Universally-optimal distributed shortest paths and transshipment via graph-based ℓ_1-oblivious routing. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2549–2579. SIAM, 2022. doi:10.1137/1.9781611977073.100.

Byzantine Connectivity Testing in the Congested Clique

John Augustine
Department of Computer Science and Engineering,
Indian Institute of Technology Madras, Chennai, India

Anisur Rahaman Molla
Indian Statistical Institute, Kolkata, India

Gopal Pandurangan
Department of Computer Science, University of Houston, TX, USA

Yadu Vasudev
Department of Computer Science and Engineering,
Indian Institute of Technology Madras, Chennai, India

Abstract

We initiate the study of distributed graph algorithms under the presence of Byzantine nodes. We consider the fundamental problem of testing the connectivity of a graph in the congested clique model in a Byzantine setting. We are given a n-vertex (arbitrary) graph G embedded in a n-node congested clique where an arbitrary subset of B nodes of the clique of size up to $(1/3 - \varepsilon)n$ (for any arbitrary small constant $\varepsilon > 0$) can be Byzantine. We consider the *full information* model where Byzantine nodes can behave arbitrarily, collude with each other, and have unlimited computational power and full knowledge of the states and actions of the honest nodes, including random choices made up to the current round.

Our main result is an efficient randomized distributed algorithm that is able to correctly distinguish between two contrasting cases: (1) the graph $G \setminus B$ (i.e., the graph induced by the removal of the vertices assigned to the Byzantine nodes in the clique) is connected or (2) the graph G is far from connected, i.e., it has at least $2|B| + 1$ connected components. Our algorithm runs in $O(\operatorname{polylog} n)$ rounds in the congested clique model and guarantees that all honest nodes will decide on the correct case with high probability. Since Byzantine nodes can lie about the vertices assigned to them, we show that this is essentially the best possible that can be done by any algorithm. Our result can be viewed also in the spirit of property testing, where our algorithm is able to distinguish between two contrasting cases while giving no guarantees if the graph falls in the grey area (i.e., neither of the cases occur).

Our work is a step towards robust and secure distributed graph computation that can output meaningful results even in the presence of a large number of faulty or malicious nodes.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Mathematics of computing → Probabilistic algorithms; Mathematics of computing → Discrete mathematics

Keywords and phrases Byzantine protocols, distributed graph algorithms, congested clique, graph connectivity, fault-tolerant computation, randomized algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.7

Funding *John Augustine*: Supported in part by Extra-Mural Research Grant (file number EMR/2016/003016), MATRICS grant (file number MTR/2018/001198), and VAJRA faculty program funded by SERB, Government of India; also supported by the potential Centre of Excellence in Cryptography Cybersecurity and Distributed Trust (CCD) under the IIT Madras Institute of Eminence scheme.
Anisur Rahaman Molla: Research supported in part by DST Inspire Faculty research grant DST/INSPIRE/04/2015/002801 and ISI DCSW/TAC Project (file number E5412).
Gopal Pandurangan: This work was supported in part by NSF grants CCF-1717075, CCF-1540512, IIS-1633720, BSF grant 2016419, and by the VAJRA faculty program of the Government of India.

© John Augustine, Anisur Rahaman Molla, Gopal Pandurangan, and Yadu Vasudev;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 7; pp. 7:1–7:21
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Computation in the presence of faulty or malicious (also called *Byzantine* nodes) is a central issue in distributed computing. Indeed, since the introduction of the Byzantine agreement and broadcast problems in the early 1980s by Lamport, Pease, and Shostak [30, 24], Byzantine protocols have been studied extensively for the last four decades (see e.g, [10, 4, 21, 20, 2] and the references therein). These protocols have focused mainly on addressing fundamental, but "primitive" computational tasks such as agreement, leader election, broadcast etc in complete networks. Some of these protocols work in the CONGEST model as well (for e.g., the protocols of [21, 2]) where messages are of small size, i.e., polylog n size in networks of size n and still guarantee fast algorithms, running in polylog n rounds. Hence these Byzantine protocols solve these fundamental tasks of agreement and leader election in complete networks in the CONGEST model (i.e., using only small-sized messages).

In this paper we take a step towards addressing more complex computational tasks under presence of a large number of Byzantine nodes. In this direction, we initiate the study of distributed graph algorithms under the presence of Byzantine nodes.

We focus on the fundamental problem of testing the connectivity of a graph in the *congested clique* model in a *full information* Byzantine setting. Graph connectivity and related problems such as minimum spanning tree (MST) have been studied extensively in the congested clique model in the last decade or so (see e.g., [27, 17, 13, 18]). In this model, we have a network that is a clique on n nodes and an (arbitrary) input graph G having n vertices that is embedded in the clique (with each vertex of G mapped to a node of the clique). Throughout we use "nodes" to denote the processors of the congested clique and "vertices" to denote the vertices of the input graph G. Connectivity (as well as MST) of the input graph G can be solved very efficiently in the congested clique, i.e., in $O(\log n)$ rounds [31]. In fact, after a long line of research, it is now known that these problems can be solved even faster, i.e., in $O(1)$ rounds [18]. All known results in the congested clique (or in the closely related k-machine model [22, 23, 14, 28, 29] or in the Massively Parallel Computing (MPC) model [3, 25, 19, 1] used in distributed large-scale graph computations) assume that all processors are honest and faithfully participate in the computation. However, in many applications, e.g., in distributed big data computations by a large network of processors, some of the processors can be faulty or even malicious.[1] Motivated by such scenarios, we assume that an arbitrary subset of B nodes of the clique of size up to $(\frac{1}{3} - \varepsilon)n$ (for any arbitrary small constant $\varepsilon > 0$) can be *Byzantine*. We assume the *full information* model where Byzantine nodes can behave arbitrarily, collude with each other, and have unlimited computational power and full knowledge of the states and actions of the honest nodes, including random choices made up to the current round. Our goal is to study how meaningful graph computations can be accomplished efficiently (say, running in a small number of rounds) even in the presence of a large number of Byzantine nodes.

It is not a priori clear how to do meaningful graph computation under a large number of Byzantine nodes. We note that Byzantine nodes can arbitrarily deviate from the correct protocol. In particular, they can lie about the edges of G incident on themselves. For example, a Byzantine node can say that an edge incident on itself is not present to some honest nodes, while saying the contrary to some other honest nodes. They can also create

[1] The situation in the MPC or the k-machine model is harder compared to the congested clique model since the number of processors (some of which can be faulty) is much smaller than the number of nodes and thus each processor is responsible for many nodes in the input graph. This setting is left for future work, while the congested clique setting addressed here is a first step in this direction.

fake edges that are not originally present in G and selectively lie about them to honest nodes. Despite such malicious behavior we show that one can design efficient protocols such that all honest nodes end up computing non-trivial meaningful results.

Our main contribution is an efficient randomized Byzantine distributed protocol that is able to correctly distinguish between two contrasting cases: (1) the graph $G \setminus B$ (i.e., the input graph induced by the removal of the vertices assigned to the Byzantine nodes in the clique) is connected or (2) the graph G is far from connected, i.e., it has at least $2|B| + 1$ connected components.[2]

Since Byzantine nodes can lie about the vertices assigned to them, we show that the above gap between the two cases is essentially the best possible that can be done by any algorithm. Our algorithm is efficient in the sense that it takes only $O(\text{polylog}\, n)$ rounds in the n-node congested clique and guarantees that *all* honest nodes will decide on the correct case with high probability. (The exact power of the $\text{polylog}\, n$ depends on Byzantine agreement and committee election protocols used in prior works that also takes $\text{polylog}\, n$ rounds (cf. Section 2.2).) Our focus in this paper, as in prior works on agreement and committee election [21, 8]), is to show that the more challenging problem of Byzantine connectivity testing can also be accomplished efficiently, i.e., in $\text{polylog}\, n$ rounds.

Our result can be viewed also in the spirit of *property testing*. In the property testing model ([16, 32]) one has query access to an input, say a graph. The goal is to test whether the input has a property, or is far (in an appropriately defined way) from the property by querying a small part of the input. Property testing algorithms in this setting has been studied for a variety of problems (see [15] for a detailed exposition). Property testing algorithms in the CONGEST model was introduced by Censor-Hillel et al. ([6]), and algorithms for properties like connectivity, cycle-freeness, bipartiteness, planarity have been studied ([9, 11, 26]). Our current work extends this to testing connectivity in the presence of Byzantine nodes. Our algorithm checks if the input graph G with $B \subseteq V$ Byzantine nodes is far from connected or whether $G \setminus B$ is connected; we will precisely define the notion of being far from connected momentarily.

1.1 Model and Problem Statement

We assume the well-studied synchronous congested clique model consisting of n nodes (representing processors or computing devices) with each node identified by a unique ID from $\{1, 2, \ldots, n\} = [n]$. (The choice of $[n]$ as the space of IDs is without loss of generality with respect to the usual assumption that IDs are from a $\text{poly}(n)$ space because nodes know each others' IDs and can use their ordinal numbers in the sorted ordering of IDs instead of actual IDs.) Communication happens over synchronized rounds, where in each round a node can communicate with any of the other $n - 1$ nodes by sending a message of $O(\log n)$ bits (i.e., the CONGEST model). We are given an arbitrary (input) graph G which is embedded in the clique and is known only locally to the nodes in the network. Every vertex in G is mapped to a node (processor) in the clique and each clique node has knowledge of the edges incident to the vertex of G mapped to it. We assume that an arbitrary subset of B nodes of the clique of size up to $(\frac{1}{3} - \varepsilon)n$ (for any arbitrary small constant $\varepsilon > 0$) can be *Byzantine*. We assume that Byzantine nodes can behave arbitrarily, collude with each other, and have

[2] It might seem a bit unnatural that in the problem definition the yes and no instances consider two different graphs: G and $G \setminus B$. However, we cannot say anything about G itself – whether it is connected or not – (as is usually the case in a non-Byzantine setting), since even the presence of one Byzantine node can lead to both possibilities.

unlimited computational power and full knowledge of the states and actions of the honest nodes, including random choices made up to the current round. This is the so-called *full information* model. We note that the Byzantine nodes in the clique are chosen after the assignment of the graph vertices to the clique.

Our goal is to test properties of the input graph G despite the presence of Byzantine nodes. The complexity measure of interest is the number of rounds to decide if G has the property. We would like to design efficient algorithms, i.e., running in a small number of rounds. In this paper, we describe an algorithm to test connectivity of a graph G in the presence of Byzantine nodes. We would like *all* honest nodes to output a common "meaningful" decision which we define next.

Since Byzantine nodes can lie about the edges incident on them, it is not possible to test connectivity in the usual sense. For example, assume that G has a cut edge and both the endpoints are assigned to Byzantine nodes. Then the Byzantine node can lie about the presence of this edge to other (honest) nodes. Hence honest nodes may conclude that the graph is not connected.[3] More generally, there is no way to verify whether the subgraph of G induced on the set of Byzantine nodes is indeed connected or not. This example, motivates that we have to relax the notion of testing connectivity as follows.

The notion of connectivity that we would like to solve is the following. Let b, be the (upper bound) number of Byzantine nodes.[4] Our goal is to test whether $G \setminus B$ is connected or G is "far from connected." We say that G is *f-far from connected*, if one must add at least f edges to G in order to make it connected (in other words, there are at least $f+1$ components in G). All honest (good) nodes must report correctly when $G \setminus B$ is connected (output TRUE) or G is f-far from connected where $f = 2b$ (output FALSE). If neither of these hold, then in this case, all honest nodes can output either TRUE or FALSE, but they should output a common decision value. We will show in Theorem 3 that this gap is essentially the best possible for any algorithm to distinguish.

While the above gap is the best possible, it helps one to distinguish between two reasonably contrasting cases. If the input graph G is well connected, say for example, it is $b+1$-connected, then the algorithm will correctly output that $G \setminus B$ is connected, since no matter which vertices are assigned to Byzantine nodes, deletion of those vertices will still leave the graph connected. On the other hand, if G far from connected, i.e., it has lot of connected components (at least $2b+1$ of them), then again, no matter which vertices are assigned to the Byzantine nodes, then the algorithm will correctly identify that G is far from connected. If neither of these scenarios occur, then the algorithm will output an arbitrary value (but consistent across all honest nodes).

1.2 Our Results

Our main result is the following theorem.

▶ **Theorem 1.** *Given a graph G embedded in an n-node congested clique, out of which an (arbitrary) subset of B nodes ($|B| < (1/3-\epsilon)n$), for any arbitrary small constant $\varepsilon > 0$, are Byzantine, we present a randomized Byzantine protocol, that with high probability[5], runs in $O(\text{polylog}\, n)$ number of rounds and outputs:*

[3] Note that the two Byzantine nodes assigned to the endpoints of the cut edge may lie to some honest nodes and not to others to prevent the honest nodes from reaching a common decision which is required; this is also an issue that one has to contend with.

[4] We assume that honest nodes have knowledge of this upper bound on the number of Byzantine nodes. Note that, in general, this is the best possible, since some Byzantine nodes can act as good nodes throughout the protocol.

[5] Throughout, "with high probability" means "with probability at least $1-1/n^c$" for some constant $c > 0$.

- *TRUE if $G \setminus B$ is connected;*
- *FALSE if G is $2|B|$-far from connected;*
- *TRUE or FALSE if neither of the above hold.*

All honest nodes output the same answer with high probability.

We also present a simple and deterministic solution. The result is stated in the following theorem. We defer a proof of this theorem to an extended version of this paper.

▶ **Theorem 2.** *Given a graph G embedded in an n-node congested clique, out of which an (arbitrary) subset of B nodes are Byzantine, where $|B| < n/3$. There is a deterministic algorithm that tests the connectivity of G in $O(n)$ rounds and outputs:*

- *TRUE if $G \setminus B$ is connected;*
- *FALSE if G is $2|B|$-far from connected;*
- *TRUE or FALSE if neither of the above hold.*

We note that the usual bound of $|B| < n/3$ Byzantine nodes in the full information model [30] for agreement holds here as well.

The gap in between the two extreme cases, i.e., whether $G \setminus B$ is connected or if it is $2|B|$-far from connected is the best possible that one can achieve in the presence of B Byzantine nodes. We defer the proof to the appendix of this paper.

▶ **Theorem 3.** *There is no algorithm that is guaranteed to distinguish whether a graph G embedded in an n-node congested clique containing a set B of b Byzantine nodes is f-far from being connected, or whether $G \setminus B$ is connected with probability at least $1/2 + \epsilon$ (for any fixed $\epsilon > 0$), unless $f \geq 2b$.*

1.3 Technical Challenges and Overview

The main difficulty in designing a protocol as claimed in Theorem 1 is correctly distinguishing the two cases or conclude it is in the gray area. If there are no Byzantine nodes, then one can simply run a distributed connectivity algorithm, in particular, a distributed minimum spanning tree (MST) algorithm (assuming all weights are 1) such as the Gallagher-Humblet-Spira (GHS) algorithm [12] – which is essentially the distributed Boruvka algorithm – on the congested clique (see e.g., [31]). This algorithm can be easily implemented to run in $O(\log n)$ rounds (deterministically) in the congested clique. However, in the presence of Byzantine nodes, this algorithm does not work.

There are two main challenges. First is to ensure that the MST algorithm correctly operates in a consistent manner despite the Byzantine nodes lying (selectively as well) about the presence or absence of edges incident to the vertices assigned to them. Second is to ensure that the algorithm operates fast, i.e., in $O(\text{polylog}\, n)$ rounds in the congested clique.

To illustrate the difficulties, as a warm up, we outline an algorithm (the detailed algorithm and its proof appears in the full version of the paper) that correctly solves the problem (i.e., it outputs correctly TRUE or FALSE depending on the cases), but takes polynomial, i.e., $O(n)$ rounds. The algorithm is deterministic. The high-level idea behind this algorithm is as follows. Each node acts as a leader and verifies all the edges of the graph. Byzantine nodes can try to foil the above algorithm by suggesting fake edges, i.e., edges that don't exist. This can be done selectively by sending different information to different honest nodes. Byzantine nodes can also reject edges that are suggested by good nodes. To overcome this, an honest node needs to verify if an edge is valid by querying both its endpoints; if both endpoints validate the edge then it is accepted; otherwise it is rejected. We show that adding

a validated or qualified edge is fine for the correctness of the algorithm; but this is not true for non-validated edges and hence they should not be added. At the end of the computation, an honest node will have all the validated edges. We show that an honest node can decide correctly by checking whether the *largest connected component* has at least $n - b$ nodes.

The above gives a fairly straightforward algorithm, though different honest nodes might decide on different answers (because of selective lying by Byzantine nodes). However, we show that if the input graph G falls in one of the cases, then all honest nodes will decide on the same answer. However, if the input graph falls in the gray area, then different honest nodes can end up with different answers. However, one can resolve this, by performing an efficient Byzantine agreement protocol. To keep the solution deterministic, we use a deterministic agreement protocol by Dolev et al. [8].

A main drawback of the above approach is that, validating all the edges can cause lot of congestion. For example, it can happen that $\Theta(n)$ edges have a common endpoint and this information has to be conveyed to every honest node which verifies it. This takes at least $\Theta(n)$ rounds.

Our main contribution is a significantly faster algorithm that runs in $O(\text{polylog}\, n)$ rounds that is able to avoid congestion and still correctly output the desired answer. At the beginning of the algorithm, we elect a *leader committee* of $O(\log n)$ leaders. The key technical part of the algorithm is for each one of the leaders to decide YES/NO. Then the $O(\log n)$ leaders will do Byzantine agreement among themselves to decide the final output (that all honest nodes can sample). Each leader implements the distributed version of Boruvka's algorithm ([12]) on the graph, but instead of each node running the algorithm, we create committees of nodes[6], and delegate the algorithm to these committees. The crucial technical ingredient is a way to construct $O(\log n)$-sized committees for each of the nodes such that the fraction of committee members that are honest is proportional to the actual fraction of honest nodes in the network. The leader samples $O(\log n)$ many hash functions which are broadcast to every node in the network. The honest nodes can use these to compute the committee pertaining to each node in the network. This makes the committees common knowledge among the nodes in the network, and prevents any tampering by Byzantine nodes. Furthermore, our construction ensures that each of the committees have a consistent view of the edges in the graph. We will then run the algorithm constructing the spanning tree via these committees.

2 Preliminaries

2.1 Tail inequalities and hash functions with limited independence

We make use of hash functions with limited independence to save messages (and hence avoid congestion). These hash functions use c-wise independence and hence we use the following tail inequalities and properties of such hash functions.
The following tail inequalities are from [33].

▶ **Lemma 4.** *Let $c \geq 4$ be an even integer. Suppose $Z_1, Z_2, \ldots, Z_t$ are c-wise independent random variables taking values in $[0,1]$. Let $Z = \sum_{i=1}^{t} Z_i$ and $\mu = \mathbb{E}[Z]$, and let $\lambda > 0$. Then,*

$$Pr[|Z - \mu| \geq \lambda] \leq 2\left(\frac{ct}{\lambda^2}\right)^{c/2}.$$

[6] The idea of communicating using committees has been used before, see e.g., [5, 20]. Note that these "node" committees are different from the leader committee, elected once at the beginning.

▶ **Lemma 5.** *Suppose that X is the summation of n 0-1 random variables $X_1, X_2, \ldots, X_n$, each with mean p. Let μ satisfy $\mu \geq \mathbb{E}[X] = np$. If the X_is are $\mu\delta$-wise independent, then for every $\delta \geq 1$*

$$\mathbb{P}[X \geq (1+\delta)\mu] \leq exp(-\delta\mu/3\}).$$

The following is Definition 7 in [7].

▶ **Definition 6.** *For N, L, $c \in \mathbb{N}$, such that $c \leq N$, a family of functions $\mathcal{H} = \{h : [N] \to [L]\}$ is c-wise independent if for all distinct $x_1, x_2, \ldots, x_c \in [N]$, the random variables $h(x_1), h(x_2), \ldots, h(x_c)$ are independent and uniformly distributed in $[L]$ when h is chosen uniformly at random from H.*

The following lemma appears as Corollary 3.34 in [34].

▶ **Lemma 7.** *For every a, b, c, there is a family of c-wise independent hash functions $\mathcal{H} = \{h : \{0,1\}^a \to \{0,1\}^b\}$ such that choosing a random function from $\mathcal{H}$ takes $c \cdot \max\{a, b\}$ random bits, and evaluating a function from $\mathcal{H}$ takes poly(a, b, c) computation.*

2.2 Byzantine agreement and committee election

As a subroutine in our protocols, we utilize the following results from King et al. [21] that gives efficient protocols (running in $O(\text{polylog}\, n)$ rounds) for Byzantine agreement and committee election in a n-node complete network (i.e., congested clique) that can tolerate up to $(1/3 - \epsilon)n$ Byzantine nodes in the full information model. We note that the protocol of King et al. has the property that every honest node sends and processes only $O(\text{polylog}\, n)$ number of bits throughout the course of the algorithm. This constrains them to achieving only almost-everywhere agreement where only $1 - o(1)$ fraction of the honest nodes agree. If we allow each node of the clique to send $O(\log n)$ bits per clique edge per round, then one can achieve everywhere agreement (i.e., all honest nodes agree) in a straightforward manner [21]. Similarly, in the committee election problem, all honest nodes can be informed of the identities of the committee nodes.

For the sake of completeness, we first define Byzantine agreement and Byzantine committee election.

In *Byzantine (everywhere) agreement*, starting with each node having an input value (0 or 1), the goal is for all honest nodes to agree on a common value, which should also be an input value of some honest node.

In *Byzantine committee election*, the goal is to elect a committee of size $\Theta(\log n)$ such that with high probability, the committee consists of at least $\frac{2}{3}$ fraction of honest processors.

▶ **Theorem 8** (Byzantine committee election [21])**.** *Consider a n-node congested clique model where up to $(1/3 - \epsilon)n$ (for any arbitrary small constant $\varepsilon > 0$) of the nodes can be Byzantine in the full information model. Then there is a randomized protocol that elects a committee of size $\Theta(\log n)$ that, with high probability, contains at least 2/3-fraction of honest nodes in the committee. The protocol runs in $O(\text{polylog}\, n)$ rounds and all honest nodes in the network know the identities of the committee members.*

▶ **Theorem 9** (Randomized Byzantine everywhere agreement [21])**.** *Consider a n-node congested clique model where up to $(1/3 - \epsilon)n$ (for any arbitrary small constant $\varepsilon > 0$) of the nodes can be Byzantine in the full information model. There is a randomized protocol that, with high probability, solves Byzantine agreement such that all honest nodes in the network agree on a common value (which will be the input value of a honest node) and runs in $O(\text{polylog}\, n)$ rounds.*

We also use a *deterministic* protocol that solves Byzantine everywhere agreement that takes linear (in the size of the network) number of rounds. We use this protocol only on a $O(\log n)$-size network (i.e., a committee of size $O(\log n)$) and hence the time taken is $O(\log n)$ rounds.

▶ **Theorem 10** (Deterministic Byzantine Everywhere Agreement [8])**.** *Consider a k-node congested clique model where fewer than $k/3$ of the nodes can be Byzantine in the full information model. There is a protocol that deterministically solves Byzantine agreement such that all honest nodes in the network agree on a common value (which will be the input value of a honest node) and runs in $O(k)$ number of rounds.*

3 An $O(\text{polylog}\, n)$-round Algorithm

In this section we give a $O(\text{polylog}\, n)$-round algorithm for testing connectivity by building committees for each vertex in the graph such that the communication between nodes is delegated to their respective committees. We ensure that the committee members are chosen randomly so that all committees, with high probability, comprise more than 2/3-fraction of good nodes. Thus, even bad nodes are represented by good committees and this significantly limits the power of the Byzantine nodes.

We first build a leader committee L of $O(\log n)$ size so that more than a 2/3-fraction of the nodes in the committee are good; see Algorithm 1. This can be done in $O(\text{polylog}\, n)$ rounds using the protocol of [21] (cf. Theorem 8). This protocol also ensures that all honest nodes know the identities of all the committee members. Now each $\ell \in L$ acts as a leader and orchestrates the execution of distributed Boruvka algorithm and decides on the output; thus, Boruvka's algorithm is invoked independently by each $\ell \in L$. This is the key technical part of the algorithm that we explain next. Once all nodes $\ell \in L$ arrive at their respective decisions, they can perform a deterministic Byzantine agreement protocol among the committee members using the protocol of [8] (cf. Theorem 10). We will run the protocol of [8] for each node in the leader committee acting as a *transmitter*.[7] This guarantees consistent agreement among the committee nodes of the value of this transmitter node. (The running time will be $O(\log^2 n)$ rounds since one run of the transmitter protocol takes linear number of rounds in the size of the committee which is $O(\log n)$.) Since the majority of the nodes in the leader committee are honest, this also ensures that the majority answer is the correct answer agreed by all honest nodes in the committee. All other honest nodes can then query all the committee nodes and then take their majority output as their own output. The formal steps are outlined in Algorithm 1.

We now briefly comment on how each $\ell \in L$ orchestrates the execution of Boruvka's algorithm. We show that it is possible for each ℓ to construct committees of size $O(\log n)$ for each vertex with the property that if at most a β fraction of the nodes in the network are Byzantine for a fixed constant $\beta < 1/3$, then, with high probability, in every committee at most $\beta + \epsilon < 1/3$ fraction of nodes are Byzantine (for a small constant $\epsilon > 0$). Moreover, each committee $\mathsf{Comm}(v)$ learns the edges incident at the node v that it represents and can interact with committees representing v's neighbors. In other words, the committees (in a collective sense) learn a consistent view of the graph. Finally, the nodes are balanced across the committees in the sense that no node needs to be a member of more than $O(\log^2 n)$ committees. Theorem 15 proves all these properties formally. We then show how to construct the spanning tree using these committees and thus test connectivity.

---

[7] This is to ensure majority consensus, i.e., the agreed value is also the majority value among the inputs, if such a majority exists.

Algorithm 1 THE MAIN STEM.

1: Use the protocol in [21] to elect a leader committee L of size $\Theta(\log n)$. /* (cf. Theorem 8 for relevant properties.) */
2: **for** each $\ell \in L$ in parallel **do**
3: Invoke Algorithm 8 with leader ℓ.
4: /* ℓ learns either *accept* or *reject* as its outcome */
5: The nodes in L perform Byzantine agreement on their outcomes using [8] (as outlined above) to reach their *final outcomes*.
6: Each $\ell \in L$ sends their final outcome to all nodes in $[n] \setminus L$.
7: All nodes in $[n] \setminus L$ set their final outcomes to the value sent by a majority in L.

3.1 Committees Takeover

We now consider the case where a single leader ℓ is able to produce a committee structure that maps every node v in the network to its representative $\Theta(\log n)$-sized committee $\mathsf{Comm}(v)$ comprising sufficiently many good nodes. Moreover, this committee structure must be common knowledge in the sense that every node u should be able to compute $\mathsf{Comm}(v)$ for all nodes v. Each $\mathsf{Comm}(v)$ can then perform computation and communication on behalf of the node v that it represents. However, each $\mathsf{Comm}(v)$ must first learn the required information (such as the neighborhood of v). If v is Byzantine, the information $\mathsf{Comm}(v)$ learns may be incorrect and inconsistent, but the committees should ensure consistency and also ensure correctness as far as possible.

We assume the following throughout this section. Let $\beta < 1/3$ and $\varepsilon < 1/3 - \beta$ be fixed constants; thus, $\beta + \varepsilon$ is fixed and strictly below $1/3$. We assume that, out of a total of $|V| = n$ nodes, the number of Byzantine nodes is at most βn. Let η be a sufficiently large constant that depends on β and ε, and let $k = \lceil \eta \log n \rceil$. Recall that the set of nodes in the network are $[n]$. When we refer to vertex $v \in [n]$, we are referring to the vertex in G that is assigned to v. Similarly, for $u \in [n]$ and $v \in [n]$, we use $(u, v) \in E(G)$ (resp., $(u, v) \notin E(G)$) to denote that the vertices assigned to u and v form an edge (resp., do not form an edge) in G. Finally, we use $N(v)$ to denote the set $\{u \in [n] \setminus \{v\} \mid (v, u) \in E(G)\}$.

The leader ℓ uses k-wise independent hash functions (cf. Definition 6) in order to build the committee structure. From Lemma 7, we know that there is a k-wise independent family of hash functions $\mathcal{H}$ with each $h \in \mathcal{H}$ mapping $[n] \to [n]$. Moreover, picking a random h only requires $k + 2\lceil \log n \rceil \in O(\log n)$ bits. Thus, ℓ generates the required $\Theta(\log^2 n)$ random bits that can be used to pick k random hash functions $h_1, h_2, \ldots, h_k$. The leader then broadcasts those $O(\log^2 n)$ bits to all nodes in the network, thereby making the k hash functions common knowledge amongst all the nodes. The committee for each node $v \in [n]$ comprises nodes in $\{h_i(v) | 1 \le i \le k\}$, thereby making the entire committee structure common knowledge. Thus, we can summarize the properties of the committee structure as follows.

▶ **Lemma 11.** *The committee structure created by a good leader node ℓ comprises commonly known committees $\mathsf{Comm}(v)$ for all $v \in V$ such that $|\mathsf{Comm}(v)| \le k \in O(\log n)$. Moreover, the randomness in the choice of committee members ensures WHP that:*

1. *the proportion of Byzantine nodes within each committee is at most $\beta + \varepsilon + o(1) < 1/3$, and*
2. *for every $c \in V$, $C(c) \triangleq \{v | c \in \mathsf{Comm}(v)\}$, i.e., the set of nodes that c represents as a committee member, has a cardinality of at most $O(\log^2 n)$.*

Algorithm 2 COMMITTEES-LEARN-INCIDENT-EDGES.

Require: Assume a good leader ℓ is elected and known to all nodes. Let $k = \lceil \eta \log n \rceil$ for a sufficiently large but fixed η.
Ensure: For each $v \in V$, an associated committee $\mathsf{Comm}(v)$ is created. See Theorem 15 for a detailed listing of all other guarantees.

1: The leader ℓ generates $\Theta(\log n)$ bits drawn uniformly and independently at random and broadcasts them to all other nodes.
2: All nodes v use those $\Theta(\log n)$ bits to draw $h_1, h_2, \ldots, h_k$ from the k-wise independent family $\mathcal{H}$ of hash functions.
/* Thus, any node v can compute the committee $\mathsf{Comm}(u) = \{h_i(u) | 1 \leq i \leq k\}$ associated with any node u. */
3: **In parallel** $\forall v \in V$ **do**
4: Node v sends the cardinality of its neighborhood $|N(v)|$ to every $c \in \mathsf{Comm}(v)$.
5: **In parallel** $\forall c \in \mathsf{Comm}(v)$, and $\forall i \in \{1, 2, \ldots, |N(v)|\}$ **do**
6: c elects a *routing committee* $\mathsf{RouteComm}(c, v, i)$ of cardinality $\Theta(\log n)$ WHP in the following manner. For every $u \in V \setminus \{c, v\}$, $\mathbb{P}[u \in \mathsf{RouteComm}(c, v, i)] = \Theta(\frac{\log n}{n})$, independently over all u and all i.
7: c sends a request for ith neighbor of v to all members in $\mathsf{RouteComm}(c, v, i)$.
8: **In parallel** each $r \in \mathsf{RouteComm}(c, v, i)$ **do**
9: **if** $c \in \mathsf{Comm}(v)$ **then**
10: r conveys the message from c to v.
11: Let q be the number of request messages c conveyed to v through r.
12: **if** $q \in O(\log n)$ **then**
13: v responds to r's request with the ID of its ith neighbor.
14: r relays the responses back to c.
15: **End parallel**
16: Let μ be the ID reported by most nodes in $\mathsf{RouteComm}(c, v, i)$.
17: **if** a majority in $\mathsf{RouteComm}(c, v, i)$ reported μ **then**
18: c records the ith neighbor as μ.
19: **else**
20: Reject the response.
21: /* c can conclude that v is Byzantine. */
22: **End parallel**
23: **End parallel**
24: **In parallel** $\forall$ pairs $u \in V$ and $v \in V$ **do**
25: Ensure all good nodes in $\mathsf{Comm}(u)$ and $\mathsf{Comm}(v)$ have a consistent record of whether edge $(u, v) \in E(G)$. Note that if both u and v are good, then their record, in addition to being consistent, must be correct. This is achieved by calling Algorithm 6.
26: **End parallel**

In order to execute graph algorithms, we require one more crucial ingredient. For each vertex v, the members of $\mathsf{Comm}(v)$ must learn the list of neighbors of v and that procedure is specified in Algorithm 2. One may be tempted to think that this can be achieved simply by v sending its adjacency list to each $c \in \mathsf{Comm}(v)$, but this will take time proportional to v's degree, which can be linear in n leading to an algorithm that is linear in n. To sidestep this delay, we can try random routing to parallelize this process; node v communicates the

ID of each neighbor of v to each $c \in \mathsf{Comm}(v)$ through a randomly chosen intermediate node. However if the random intermediate node is Byzantine, the information may be misrepresented. To overcome this, we use randomly chosen *routing committees* as formally described in Algorithm 2. Some care must be exercised because routing committees can also contain Byzantine nodes and the information passing through them has to therefore be vetted.

▶ **Lemma 12.** *By the end of the first parallel loop in Algorithm 2 ending in line number 23, every $c \in \mathsf{Comm}(v)$ for $v \in [n]$ has learned a list of neighbors of v with the guarantee that if both c and v are good, the list matches $N(v)$ correctly. Moreover, the time taken for that first parallel loop to complete is $O(\mathrm{polylog}(n))$ WHP.*

At the end of the first parallel loop (line number 23), the nodes in the committees pertaining to two nodes u and v may not reach the same conclusion about whether $(u, v) \in E(G)$ or not. To ensure consistency and correctness (to the extent possible), we perform some agreement and consistency tie-breaks in the second parallel loop (post line number 23). To implement this second loop, we need Algorithm 6, which in turn utilizes some key primitives. The main idea is to first ensure that for each u, the committee $\mathsf{Comm}(u)$ first reaches an agreement on whether there is an edge from u to each v. Each committee therefore has to perform $n - 1$ agreements, thereby requiring a total of $n(n - 1)$ agreements. These large number of agreements are performed in parallel in $O(\mathrm{polylog}(n))$ time (WHP) using Algorithm 5.

We begin with some simpler primitives that will then be used by Algorithm 5 and Algorithm 6. Our first primitive is Algorithm 3 where each the committee members of a node u can communicate messages pertaining to all other nodes v in $O(\mathrm{polylog}(n))$ rounds.

Algorithm 3 Communication between committee members across committees. This algorithm is described from the perspective of a node c.

Require: For every $u \in V$, $c \in \mathsf{Comm}(u)$, and $v \in V$, the node c has a message $m(c, u, v)$. /* There are $O(n^2 \log n)$ such messages WHP. */

Ensure: Every $c' \in \mathsf{Comm}(v)$ must learn $m(c, u, v)$ as long as c is good.

1: **for** each $u \in C(c)$ and each $c' \in \mathsf{Comm}(v)$ **do** /* Recall that for each $c \in V$, $|C(c)| \in O(\log^2 n)$ WHP by Lemma 11. */

2: Node c sends $m(c, u, v)$ to c'.

3: Node c receives $m(c', v, u)$ sent by c'.

▶ **Lemma 13.** *Algorithm 3 terminates in $\Theta(\log^4 n)$ rounds WHP. Moreover, for every good c, all its messages are correctly conveyed (i.e., all messages are passed to the intended recipient).*

The second primitive is Algorithm 4 wherein for each u, the members of $\mathsf{Comm}(u)$ broadcast messages to each other. Specifically, each $c \in \mathsf{Comm}(u)$ has a message pertaining to the pair u and v. Thus, c is in possession of $O(n)$ messages under the role of $c \in \mathsf{Comm}(v)$; it may have other such messages as a member of other committees. All those $O(n)$ messages must reach all other $c'' \in \mathsf{Comm}(u)$. Due to the number of messages, c cannot directly send them to c''. Instead, c sends each message pertaining to the pair u and v to the members of $\mathsf{Comm}(v)$ using Algorithm 3 and then the members of $\mathsf{Comm}(v)$ echo them back to the members in $\mathsf{Comm}(u)$. When c and c'' are good, clearly, c'' will hear a majority of consistent messages correctly from c'' because the committees that are used to echo the messages are all dominated by good nodes. Furthermore, Algorithm 4 only requires $O(\log n)$ calls to Algorithm 3, thereby allowing us to conclude the following.

▶ **Lemma 14.** *For every pair u and v, every good node $c \in \mathsf{Comm}(u)$ will be able to broadcast its messages of the form $m(c,u,v)$ (see Algorithm 4) to all nodes in $\mathsf{Comm}(u)$ using Algorithm 4. Moreover, its complexity is $O(\log^5 n)$ rounds WHP.*

Algorithm 4 Broadcasting within each committee.

Require: For every $u \in V$, $c \in \mathsf{Comm}(u)$, and $v \in V$, the node c has a message $m(c,u,v)$. /* As before, there are $O(n^2 \log n)$ such messages WHP. */

Ensure: For all u, every $c' \in \mathsf{Comm}(u)$ must learn $m(c,u,v)$ ($\forall c \in \mathsf{Comm}(u)$) correctly as long as c is good. /* Note that each c will possess $O(n \log n)$ messages of the form $m(c,\cdot,\cdot)$ and it must learn $O(n \log n)$ messages. */

1: /* To describe this algorithm, we focus on a pair u and v and a $c \in \mathsf{Comm}(u)$ and trace each copy of $m(c,u,v)$ as it makes its way to all other $c'' \in \mathsf{Comm}(u)$. This process must be executed in parallel for every u and v and every $c \in \mathsf{Comm}(u)$. For this reason, note that $m(c,u,v)$ cannot be directly sent to all $c'' \in \mathsf{Comm}(u)$ because there are a linear in n number of messages of the form $m(c,u,\cdot)$ that must reach c''. */
2: The message $m(c,u,v)$ reaches each $c' \in \mathsf{Comm}(v)$. This requires one invocation of Algorithm 3.
3: Each $c' \in \mathsf{Comm}(v)$ now sends $m(c,u,v)$ to each $c'' \in \mathsf{Comm}(u)$. This requires $O(\log n)$ invocations of Algorithm 3.
/* Each $c'' \in \mathsf{Comm}(u)$ has at most k copies of $m(c,u,v)$ sent by each $c' \in \mathsf{Comm}(v)$. */
4: **for** each $c'' \in \mathsf{Comm}(u)$ **do**
5: **if** at least $\lceil (k+1)/2 \rceil$ copies of $m(c,u,v)$ are identical **then**
6: c'' accepts the majority copy of $m(c,u,v)$.
7: **else**
8: Reject the message $m(c,u,v)$.

Having established the two primitives Algorithm 3 and Algorithm 4, we now present Algorithm 5 and Algorithm 6.

Algorithm 5 Parallel Agreement.

Require: For every $u \in V$, $c \in \mathsf{Comm}(u)$, and $v \in V$, the node c has a bit $b(c,u,v)$.
1: /* As before, there are $O(n^2 \log n)$ such bits WHP. */
Ensure: For each u and each v, the members of $\mathsf{Comm}(u)$ must reach an agreement over the bits $b(c,u,v)$ over all $c \in \mathsf{Comm}(u)$.
2: **In parallel** each pair u and v **do**
3: The members of $\mathsf{Comm}(u)$ implement an $O(\text{polylog}\, n)$ round Byzantine Agreement algorithm [8] using Algorithm 4 to communicate with each other.
4: **End parallel**

▶ **Theorem 15.** *Consider the protocol described in Algorithm 2. Consider any constant $\beta < 1/3$ and a sufficiently small $\epsilon \in (0, 1/3-\beta)$ to be fixed constants. We assume that the leader ℓ is good and the number of Byzantine nodes $B < \beta n$. The following hold with high probability.*

1. *Algorithm 2 ensures that every node $v \in V$ is represented by a committee $\mathsf{Comm}(v) \subseteq V$ of cardinality $\Theta(\log n)$. Each node participates in at most $O(\log^2 n)$ committees.*
2. *This committee structure is common knowledge amongst all nodes.*
3. *Byzantine nodes amount to a fraction of at most $\beta+\epsilon+o(1) < 1/3$ within each committee.*

Algorithm 6 Ensuring a consistent view of edges.

Require: For every $u \in V$, $c \in \mathsf{Comm}(u)$, and $v \in V$, the node c has a bit $b(c, u, v)$. If u is good then $b(c, u, v) = 1$ iff $(u, v) \in E(G)$. /* Note that c can be Byzantine and may misrepresent its bit values. */

Ensure: For every $u \in V$ and $v \in V$, the members of $\mathsf{Comm}(u)$ must reach an agreement on whether (u, v) exists. Moreover, it should be consistent with the agreement reached by $\mathsf{Comm}(v)$ regarding edge (u, v), i.e., $\mathsf{Comm}(u)$ must agree that the edge (u, v) exists iff $\mathsf{Comm}(v)$ agrees that (u, v) exists. Finally, if both u and v are good, then both $\mathsf{Comm}(u)$ and $\mathsf{Comm}(v)$ must agree that edge (u, v) exists iff $(u, v) \in E(G)$.

1: Use Algorithm 5 to ensure that for every u and every v, the members in $\mathsf{Comm}(u)$ reach an agreement (using their $b(c, u, v)$ bits). Denote the agreed bit as $a(u, v)$. Each $c \in \mathsf{Comm}(u)$ sets $a(c, u, v) \leftarrow a(u, v)$.

2: Use Algorithm 3 to ensure that for every u and every v, the members of $\mathsf{Comm}(v)$ learn $a(c, u, v)$ for every $c \in \mathsf{Comm}(u)$. /* This also means that the members of $\mathsf{Comm}(u)$ learn $a(c', v, u)$ for every $c' \in \mathsf{Comm}(v)$. */

3: For every u and every v, the members of $\mathsf{Comm}(u)$ learn $a(v, u)$ by taking majority over all $a(\cdot, v, u)$ values received in the previous step.

4: For every u and every v, the members of $\mathsf{Comm}(u)$ set their final agreement bit $f(u, v)$ to 1 iff both $a(u, v) = 1$ and $a(v, u) = 1$.

---

4. *For every pair of nodes u and v, $\mathsf{Comm}(u)$ has reached an agreement about whether $(u, v) \in E(G)$. Note that $\mathsf{Comm}(v)$ has also reached its own agreement. The agreements satisfy the following two properties.*

Consistency. *$\mathsf{Comm}(u)$ reaches the agreement that $(u, v) \in E(G)$ iff $\mathsf{Comm}(v)$ reaches the agreement that $(u, v) \in E(G)$.*

Correctness. *If u is good and $(u, v) \notin E(G)$, the committees reach the agreement that $(u, v) \notin E(G)$. Furthermore, if both u and v are good nodes, then, their committees reach the agreement that $(u, v) \in E(G)$ iff $(u, v) \in E(G)$.*

5. *Finally, the protocol terminates in $O(\mathrm{polylog}\, n)$ rounds.*

Proof. Item (i) follows immediately from Lemma 11. Item (ii) follows from Algorithm 2 where the leader sends the $O(\log^2 n)$ bits to all the nodes and the nodes can then compute the committee structure locally at all nodes. Item (iii) is proved in item (i) of Lemma 11. Item (v) follows from Lemma 12, Lemma 13, Lemma 14, and the fact that Algorithm 6 only requires $O(\log n)$ invocations of Algorithm 3 and Algorithm 4.

For item (iv), let us focus our attention on Algorithm 6. Note that for each u the associated $\mathsf{Comm}(u)$ internally reaches agreement in Line 1 about whether $(u, v) \in E(G)$ for every other v. Then, for each pair u and v, their respective committees learn about the agreement reached by the other committee; see Line 3. In Line 4, both committees reach a consistent agreement. Moreover, they reach a common agreement that the edge $(u, v) \in E(G)$ iff both $\mathsf{Comm}(u)$ and $\mathsf{Comm}(v)$ internally reached that agreement. Thus, as long as one of them (i.e., either u or v) is good, $\mathsf{Comm}(u)$ and $\mathsf{Comm}(v)$ cannot reach a joint agreement that $(u, v) \in E(G)$ when $(u, v) \notin E(G)$. On the flip side, when both u and v are good, their committees would have reached the correct internal agreement and therefore the common agreement will also be correct. ◀

3.2 Constructing a spanning tree using committees

As mentioned earlier, we will implement a distributed version of Boruvka's algorithm by delegating the work done by each of the vertices in the network to their corresponding committees. We will start by describing an implementation of Boruvka's algorithm in the congested clique in the absence of Byzantine nodes.

The algorithm proceeds in phases, and in phase i, we have a collection of fragments $F_1, F_2, \ldots, F_r$ that are disjoint connected components of the underlying graph. Each fragment F_i has an associated fragment identifier, which is the name of one of the vertices in the fragment. We will maintain the invariant that after each phase, every vertex in the graph knows the fragment ids of every other vertex in the graph. Initially, the fragments are individual vertices, and the fragment ids are their own ids. Every vertex transmits their id to every other vertex in the clique.

Suppose that after the ith phase, the fragments are $F_1, F_2, \ldots, F_r$ and that the invariant holds. Let f_v denote the fragment id of the vertex v. We will assume that the algorithm always chooses the lexicographically smallest outgoing edge from a fragment. Now, each vertex chooses the lexicographically least u in the neighborhood of v such that $f_u \neq f_v$ and broadcasts the id of u to every vertex in the clique. Now, each node can calculate the lexicographically least outgoing edge from every fragment and hence compute the new fragments and their ids. Since the number of fragments reduce by a factor of at least two every step, the number of rounds is $O(\log n)$. Notice that this algorithm is deterministic, and ends when the graph has been partitioned into connected components. Now we will see how to implement this algorithm in the presence of Byzantine nodes using committees as described in the previous subsection.

Algorithm 7 SINGLE PHASE OF DISTRIBUTED BORUVKA.

Require: Graph partitioned into fragments $F_1, F_2, \ldots, F_r$, with a fragment id associated with each fragment. Conditions of Theorem 15 hold. For each u, $\mathsf{Comm}(u)$ knows the fragment id of u which we will denote by f_u.

Ensure: Find one inter-fragment edge per fragment, and update fragment ids

1: **In parallel** $\forall c \in V$ **do**
2: $\forall u$ such that $c \in \mathsf{Comm}(u)$, send $(id(u), f_u)$ to every vertex in the clique
3: **End parallel**
4: **In parallel** $\forall c \in V$ **do**
5: $\forall u$ such that $c \in \mathsf{Comm}(u)$, find u with smallest id such that $f_u \neq f_v$ and send $(id(u), id(v))$ to every vertex in the clique.
6: **End parallel**

Correctness and Complexity. From Lemma 11, we know that, with high probability, at least half of the nodes in a committee is good. Hence, for each u, every vertex v can compute the committee of u, $\mathsf{Comm}(u)$ and use the majority value of its messages from $\mathsf{Comm}(u)$ to compute f_u. Similarly, for each u, majority of the vertices in $\mathsf{Comm}(u)$ will correctly calculate its smallest outgoing edge. Thus each vertex v, can use the majority answer from $\mathsf{Comm}(u)$ to find the outgoing edge from u. Thus, each vertex can compute the correct outgoing edge from a fragment and update the fragment ids accordingly. By Lemma 11, we know that, with high probability, each node c is present in the committee of $O(\log n)$ other vertices. Hence the round complexity of Algorithm 7 is $O(\log^2 n)$.

We now describe the final algorithm.

Algorithm 8 CONNECTIVITY TESTING.

Require: Graph G in the congested clique model, and a node ℓ designated as leader known to every vertex in the graph

Ensure: Leader ℓ outputs accept (TRUE) if $G \setminus B$ is connected, and outputs reject (FALSE) if G is $2|B|+1$-far from being connected.

1: Use Algorithm 2 to obtain committees for each of the nodes in the graph that satisfies the conditions of Theorem 15.
2: Set every vertex u as a singleton fragment with fragment id as the vertex id.
3: Each node executes Algorithm 7 until there are no inter-fragment edges.
4: Leader ℓ accepts if there is a component of size at least $n - |B|$. Else ℓ rejects.

The following two lemmas prove the correctness of the algorithm. Lemma 16 shows that if indeed $G \setminus B$ is connected, then this will be observed by the leader with high probability, and Lemma 17 shows that the leader will reject, with high probability, if G is $2|B|$-far from being connected. If neither of the two conditions hold, then all the honest nodes converges on a consistent answer even though we can give no guarantee on what that answer will be.

▶ **Lemma 16.** *If the graph $G \setminus B$ is connected, then the leader returns accept with high probability in Algorithm 8.*

▶ **Lemma 17.** *Let $\beta < 1/3$ be a constant such that $|B| = \beta n$. If the graph G is $2|B|$-far from being connected, then the leader returns reject with high probability in Algorithm 8.*

We now show how to combine the results for the proof of Theorem 1.

Proof of Theorem 1. First, we will elect a leader committee of $\Theta(\log n)$ nodes such that at least 2/3-fraction of these nodes are honest. For this we use Theorem 8. Now, each leader node in this committee runs Algorithm 8 to check the connectivity of the graph. From Lemmas 17 and 16, we can conclude that if the leader node is honest, then it obtains the correct answer with high probability. To obtain the final answer we use the consensus algorithm of [8] (Theorem 10), which guarantees that the consensus obtained by the leader committee is the value of the transmitter node. Since each honest node has the wrong answer with probability at most $1/n^c$, for some constant c, by a union bound we can conclude that with high probability all honest nodes have the correct answer. Thus, repeating the consensus algorithm of [8] with each node acting as the transmitter ensures that for the majority of the values are the correct answer with high probability. ◀

4 Conclusion and Future Work

In this paper, we take a step towards robust and secure distributed graph computation by a network of nodes that can output meaningful results even in the presence of a large number of faulty or malicious nodes. This can be relevant in distributed big data applications where a large number of processors operate on a (large-scale) input and some of the processors can be faulty or malicious. We address the fundamental problem of connectivity and show that non-trivial meaningful computation can be performed in an efficient manner even in the presence of a large number of Byzantine nodes. One of the key aspects of studying graph problems in this Byzantine fault-tolerant setting is arriving at the correct problem formulation. One must contend with the fact that Byzantine nodes will enforce a "gap" between two solvable ends of a spectrum of input instances. Our work shows that we can indeed formulate such gap problems in a meaningful way. The techniques used in this paper may be useful in Byzantine distributed computation of other problems in the congested clique.

Our approach can serve as a starting point for a general technique for transforming non-Byzantine congested-clique algorithms to Byzantine ones. This work which addresses the basic connectivity problem is a step towards this larger goal which is harder. For example, generalizing our approach to even the closely related MST problem (in a weighted graph) is not immediate as it is not clear what a meaningful output for MST will be in a weighted graph in a Byzantine setting. This is an important next step and is left for future work.

Several open questions arise from our work. While the focus of this work is fast algorithms (running in polylog(n) rounds), it is also relevant to consider the message complexity of the protocol. The proposed algorithm takes $O(n^2 \operatorname{polylog}(n))$ messages in the worst case. This message complexity is essentially optimal (upto a polylog(n) factor) in the so-called KT0 model (where nodes don't have apriori information about the identities of its neighbors) since $\Omega(n^2)$ is a lower bound for connectivity testing in the congested clique even when there are no Byzantine nodes[17]. The situation is not clear if nodes have knowledge of the identities of their neighbors initially (so-called KT1 model). In this case, the question of whether $o(n^2)$ or even $O(n \operatorname{polylog} n)$ message algorithms are possible is open. Note that this is possible if there are no Byzantine nodes [17].

An interesting way to characterize performance of Byzantine protocols for more complicated problems such as connectivity is to parameterize the complexity of the performance in terms of the number of calls to Byzantine agreement which is a basic primitive. In this direction, the goal is to reduce the number of calls to agreement as much as possible.

A major next step is addressing even more challenging problems such as MST and other graph problems.

References

1. A. Andoni, Z. Song, C. Stein, Z. Wang, and P. Zhong. Parallel graph connectivity in log diameter rounds. In *Proc. IEEE FOCS*, pages 674–685, 2018.
2. John Augustine, Valerie King, Anisur Rahaman Molla, Gopal Pandurangan, and Jared Saia. Scalable and secure computation among strangers: Message-competitive byzantine protocols. In *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 31:1–31:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
3. Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *J. ACM*, 64(6):40:1–40:58, 2017.
4. Michael Ben-Or, Elan Pavlov, and Vinod Vaikuntanathan. Byzantine agreement in the full-information model in o(log n) rounds. In *Proc. of the 38th Annual ACM Symposium on Theory of Computing (STOC)*, pages 179–186, 2006.
5. Elette Boyle, Shafi Goldwasser, and Stefano Tessaro. Communication locality in secure multi-party computation - how to run sublinear algorithms in a distributed setting. In *Proc. of the 10th Theory of Cryptography Conference (TCC)*, pages 356–376, 2013.
6. Keren Censor-Hillel, Eldar Fischer, Gregory Schwartzman, and Yadu Vasudev. Fast distributed algorithms for testing graph properties. *Distributed Comput.*, 32(1):41–57, 2019. doi:10.1007/s00446-018-0324-8.
7. Artur Czumaj, Peter Davies, and Merav Parter. Simple, deterministic, constant-round coloring in the congested clique. *Proceedings of the 39th Symposium on Principles of Distributed Computing*, July 2020. doi:10.1145/3382734.3405751.
8. Danny Dolev, Michael J. Fischer, Robert J. Fowler, Nancy A. Lynch, and H. Raymond Strong. An efficient algorithm for byzantine agreement without authentication. *Inf. Control.*, 52(3):257–274, 1982.

9 Guy Even, Orr Fischer, Pierre Fraigniaud, Tzlil Gonen, Reut Levi, Moti Medina, Pedro Montealegre, Dennis Olivetti, Rotem Oshman, Ivan Rapaport, and Ioan Todinca. Three notes on distributed property testing. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 15:1–15:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPIcs.DISC.2017.15`.

10 Pesech Feldman and Silvio Micali. An optimal probabilistic protocol for synchronous byzantine agreement. *SIAM J. Comput.*, 26(4):873–933, 1997.

11 Pierre Fraigniaud and Dennis Olivetti. Distributed detection of cycles. *ACM Trans. Parallel Comput.*, 6(3):12:1–12:20, 2019. `doi:10.1145/3322811`.

12 R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, January 1983. `doi:10.1145/357195.357200`.

13 Mohsen Ghaffari and Merav Parter. MST in log-star rounds of congested clique. In George Giakkoupis, editor, *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 19–28. ACM, 2016.

14 S. Gilbert and L. Li. How fast can you update your MST. In *Proc. ACM SPAA*, pages 531–533, 2020.

15 Oded Goldreich. *Introduction to Property Testing*. Cambridge University Press, 2017. `doi:10.1017/9781108135252`.

16 Oded Goldreich, Shafi Goldwasser, and Dana Ron. Property testing and its connection to learning and approximation. *J. ACM*, 45(4):653–750, 1998. `doi:10.1145/285055.285060`.

17 James W. Hegeman, Gopal Pandurangan, Sriram V. Pemmaraju, Vivek B. Sardeshmukh, and Michele Scquizzato. Toward optimal bounds in the congested clique: Graph connectivity and MST. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 91–100. ACM, 2015.

18 Tomasz Jurdzinski and Krzysztof Nowicki. MST in $O(1)$ rounds of congested clique. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 2620–2632. SIAM, 2018.

19 H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for MapReduce. In *Proc. ACM SPAA*, pages 938–948, 2010.

20 Valerie King and Jared Saia. Breaking the $O(n^2)$ bit barrier: Scalable byzantine agreement with an adaptive adversary. *J. ACM*, 58:18:1–18:24, July 2011.

21 Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Scalable leader election. In *SODA*, pages 990–999, 2006.

22 Hartmut Klauck, Danupon Nanongkai, Gopal Pandurangan, and Peter Robinson. Distributed computation of large-scale graph problems. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 391–410. SIAM, 2015.

23 C. Konrad, S. V. Pemmaraju, T. Riaz, and P. Robinson. The complexity of symmetry breaking in massive graphs. In *Proc. DISC*, volume 146, pages 26:1–26:18, 2019.

24 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

25 S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: a method for solving graph problems in MapReduce. In *Proc. ACM SPAA*, pages 85–94, 2011. `doi:10.1145/1989493.1989505`.

26 Reut Levi, Moti Medina, and Dana Ron. Property testing of planarity in the CONGEST model. *Distributed Comput.*, 34(1):15–32, 2021. `doi:10.1007/s00446-020-00382-3`.

27 Zvi Lotker, Elan Pavlov, Boaz Patt-Shamir, and David Peleg. MST construction in o(log log n) communication rounds. In Arnold L. Rosenberg and Friedhelm Meyer auf der Heide, editors, *SPAA 2003: Proceedings of the Fifteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 7-9, 2003, San Diego, California, USA (part of FCRC 2003)*, pages 94–100. ACM, 2003.

28 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. Fast distributed algorithms for connectivity and MST in large graphs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, pages 429–438. ACM, 2016.

29 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. On the distributed complexity of large-scale graph computations. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*, pages 405–414. ACM, 2018.

30 Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

31 David Peleg. *Distributed Computing: A Locality-Sensitive Approach.* SIAM, Philadelphia, 2000.

32 Ronitt Rubinfeld and Madhu Sudan. Self-testing polynomial functions efficiently and over rational domains. In *Proceedings of the Third Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms*, pages 23–32. ACM/SIAM, 1992.

33 Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-hoeffding bounds for applications with limited independence. *SIAM J. Discret. Math.*, 8(2):223–250, 1995.

34 Salil P. Vadhan. Pseudorandomness. *Foundations and Trends in Theoretical Computer Science*, 7(1-3):1–336, 2012.

A Proof from Section 1

Proof of Theorem 3. Consider any $1 \leq b \leq n$ and $f = 2b - 1$. Suppose (for the sake of contradiction) that there is an algorithm $\mathcal{A}$ that can distinguish the following two cases with probability at least $1/2 + \epsilon$: (i) G is f-far from being connected, or (ii) $G \setminus B$ is connected. For clarity, we will use the term global algorithm to refer to the collective execution of an algorithm at all nodes and the term protocol to refer to the specific execution at an algorithm at a particular node.

We prove this theorem by constructing two graphs: (i) G_f that is f-far from being connected and (ii) G_c with a connected component of size $n - b$. We then show embeddings of these graphs on n node congested cliques with b Byzantine nodes such that the Byzantine nodes can render both embeddings indistinguishable to the global algorithm $\mathcal{A}$.

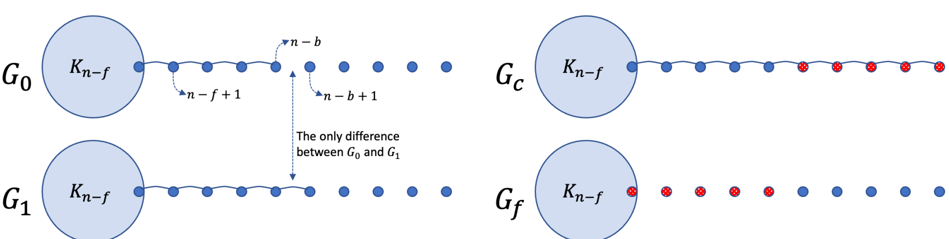


Figure 1 Graphs used in the proof of Theorem 3.

To aid us in formally showing this indistinguishability, we first construct two graphs G_0 and G_1 (that only differ in one edge) and embed them in congested cliques. See Figure 1 for schematics. Executing $\mathcal{A}$ on G_0 and G_1 will provide us with specific protocols executed by specific nodes in their respective congested cliques that can be used to precisely specify the protocols executed by Byzantine nodes in the embeddings of G_c and G_f. Both G_0 and G_1 comprise a clique of size $n - f$ (numbered 1 through $n - f$). G_0 (resp., G_1) has a "tail" comprising $b - 1$ nodes (resp., b nodes) numbered $n - f + 1$ to $n - f + b - 1$ (resp., $n - f + b$).

Specifically, the tail is a path comprising $b-1$ nodes (resp., b nodes) with one of its ends (node numbered $n-f+1$) connected to the $(n-f)$th node in the clique. Additionally, G_0 (resp., G_1) comprises $f-b+1$ (resp., $f-b$) nodes numbered $n-f+b$ to n (resp, $n-f+b+1$ to n) that are isolated. The embedding is straightforward. Vertex i is embedded into node i for all $1 \leq i \leq n$. Some arbitrary b nodes in this embedding are Byzantine. Let $\mathcal{A}_i^0$ (resp., $\mathcal{A}_i^1$) denote the protocol executed by node i in the embedding of G_0 (resp., G_1) given that i is good. Note that $\mathcal{A}$ is not required to interpret G_j, $0 \leq j \leq 1$, correctly because neither is $G_j \setminus B$ connected nor is G_j f-far from connected.

G_f comprises one clique of $n-f$ vertices (numbered 1 to $n-f$) and f isolated vertices (numbered $n-f+1$ to n); thus it is f-far from connected. See Figure 1 for a schematic representation of G_f. The clique is embedded into the first $n-f$ nodes of the congested clique and the isolated vertices are embedded into the remaining f nodes. The b Byzantine nodes are nodes $n-f, n-f+1, \ldots, n-f+b-1$. The adversary specifies the protocol executed by the Byzantine nodes i, $n-f \leq i \leq n-f+b$, to be $\mathcal{A}_i^1$. All other nodes i, by the design of $\mathcal{A}$ and the vertex embedded into them (along with incident edges matching G_g), execute their respective $\mathcal{A}_i$. Intuitively, this again produces a view of the graph that combines G_0 and G_1 with the one edge between vertices $n-b$ and $n-b+1$ under contention. The protocol $\mathcal{A}_{n-b}^1$ executed by $n-b$ will essentially operate under the existence of the edge while $\mathcal{A}_{n-b+1}$ executed by node $n-b+1$ will operate under its non-existence.

Similarly, G_c comprises one clique on $n-f$ vertices (numbered 1 to $n-f$) and a path on vertices numbered $n-f$ to n. See Figure 1 for a schematic representation of G_c. The embedding is natural, i.e., vertex i in G_c is embedded into node i in the congested clique. The b Byzantine nodes are $n-b+1$ to n and the adversary respectively executes protocol $\mathcal{A}_i^0$, $n-b+1 \leq i \leq n$, on those vertices. The rest of the vertices i, $1 \leq i \leq n-b$, by their design execute protocol $\mathcal{A}_i$. Importantly, this embedding is such that $G_c \setminus B$ is connected. Intuitively, this embedding again produces a view of the graph that combines G_0 and G_1 with the one edge between vertices $n-b$ and $n-b+1$ under contention. The protocol $\mathcal{A}_{n-b}$ executed by $n-b$ will essentially operate under the existence of the edge while $\mathcal{A}_{n-b+1}^0$ will be executed by node $n-b+1$ under its non-existence.

It is easy to verify that the protocols executed at corresponding nodes in both G_f and G_c execute the same code under the same local input. Thus, the global algorithm executed by the two embeddings of G_f and G_c are statistically identical. However, the global algorithm executed by the two embeddings must be able to distinguish with probability $1/2+\epsilon$ whether the underlying graph is G_c or G_f, thereby establishing the contradiction. ◀

B Proofs from Section 3

Proof of Lemma 11. Item (i) is easy to prove through a simple application of the Chernoff bound. Fix some node u. Recall that when $h \in \mathcal{H}$ is chosen uniformly at random, $\mathbb{P}(h(u) = v) = 1/n$. Thus, when the number of Byzantine nodes is at most βn, the $Pr(h(u)$ is Byzantine$) \leq \beta$. Since the leader chooses h_i, $1 \leq i \leq k = O(\log n)$, independently and uniformly at random from $\mathcal{H}$, we can apply Chernoff bounds to bound the number of Byzantine nodes in $\mathsf{Comm}(u)$. Let $X_i = 1$ if $h_i(u)$ is Byzantine, and 0 otherwise. Then, $X = \sum_i X_i$ will be the total number of Byzantine nodes in $\mathsf{Comm}(u)$. Clearly, $\mu = E[X] \leq \beta k$. Then, by Chernoff bounds,

$$\begin{aligned}\mathbb{P}(X \geq (\beta+\varepsilon)k) &= \mathbb{P}(X \geq (1+\varepsilon/\beta)\mu)\\ &\leq \exp\left(-\mu(\varepsilon/\beta)^2/3\right)\\ &\leq 1/\operatorname{poly}(n).\end{aligned}$$

To complete the argument that the proportion of Byzantine nodes in $\mathsf{Comm}(u)$ is at most $\beta + \varepsilon$, we also need to ensure that the committee is of cardinality at least $k - O(1)$ WHP, i.e., there aren't too many duplicated $h_i(u)$ values. This is easy to argue because, for a fixed $i \in [k]$, the probability that $h_i(u)$ is a duplicated item is at most $k/n = O(\frac{\log n}{n})$. Moreover, for $1 \leq i \leq k$, the $h_i(u)$ values are k-wise independent. Clearly then, with high probability, no more than $O(1)$ items in the committee are duplicated. Thus, we can conclude that the fraction of Byzantine nodes in the committee will be at most $(\beta+\varepsilon)k/(k-O(1)) = \beta+\varepsilon+o(1) < 1/3$ WHP when k is sufficiently large. To complete the argument, we can take the union bound over all $u \in [n]$.

To prove item (ii), we use the k-wise independence of $\mathcal{H}$ and apply Lemma 5. Let us focus on the first hash function h_1 and one node c; later we can apply the union bound over all h_i and all c. Let Y_u be 1 if $h_1(u) = c$ and 0 otherwise. If we then set $Y = \sum_u Y_u$, we get the number of committees in which c participates as the first node. Since the Y_u values are k-wise independent, we can apply Lemma 5 and get (for suitable $\delta \in \Theta(\log n)$)

$$\mathbb{P}(Y \geq (1+\delta)E[Y]) \leq \exp(-\min(k, \delta^2 t \log n)) \leq 1/\operatorname{poly}(n).$$

Item (ii) follows when we take the union bound over all h_i and all c. ◀

Proof of Lemma 12. We first bound the time and then prove correctness.

First, we need to ensure that Line number 7 takes $O(\text{polylog}(n))$ rounds WHP and this will be ensured as long as each $c \in [n]$ sends at most $O(\text{polylog}(n))$ messages. To show this, we need to bound the number of routing committees in which a node r will participate for a particular v and $c \in \mathsf{Comm}(v)$. By a straightforward application of Chernoff bounds, we get that required bound.

▶ **Lemma 18.** *Fix any v, any $c \in \mathsf{Comm}(v)$, and an arbitrary $r \in [n]$. With high probability,*

$$|\{\mathsf{RouteComm}(c, v, i) \ni r \mid i \in [N(v)]\}| \in O(\log n).$$

The requests are relayed to v and back to c only if $c \in \mathsf{Comm}(v)$. So again, the congestion in the links between c and r and between r and v will only be $O(\text{polylog}(n))$, implying that the inner parallel loop (lines 8 to 15) completes in $O(\text{polylog}(n))$ rounds. Thus, by extension, the running time for the first parallel loop is $O(\text{polylog}(n))$.

The correctness follows in a straightforward manner. If both c is good, then in expectation, fewer than β proportion of nodes will be bad in the routing committee. Thus, by a standard application of the Chernoff bound it follows that the the routing committees $\mathsf{RouteComm}(c, \cdot, \cdot)$ are all good in the sense that fewer than 1/3 proportion of nodes will be bad. Of course, bad nodes can claim to be in those routing committees. However, if v, $c \in \mathsf{Comm}(v)$, and $r \in \mathsf{RouteComm}(c, v, i)$ (for some i) are all good, then they will all follow the protocol and so each routing committee will provide a majority of correct responses. Thus, taking the union bound over all i, the proof of Lemma 12 is complete. ◀

Proof of Lemma 13. Fix a pair of nodes c and c' and consider their role as committee members. Recall that $|C(c)|$ and $|C(c')|$ are both $O(\log^2 n)$ WHP. The number of messages c sends to c' is $O(\log^4 n)$ because each $u \in C(c)$ has an associated message $m(c, u, v)$ for each $v \in C(c')$ and all these messages must traverse the link between c and c'. Moreover, no other message is passed.

The correctness follows because the messages are directly passed to all $c' \in \mathsf{Comm}(v)$. ◀

Proof of Lemma 16. If $G \setminus B$ is connected, then every non-Byzantine node is in a connected component of size at least $n - |B|$. We know that after each step of Algorithm 7, with high probability, every node knows the fragment ids of every non-Byzantine node. at the end of Algorithm 8. Therefore, at the end of the algorithm, the leader ℓ has the fragment ids of every non-Byzantine node. Since they all will lie in the same component, the leader will accept. ◀

Proof of Lemma 17. If G is $2|B|$-far from being connected, then then there are at least $2|B| + 1$ connected components in the graph. Since there are only $|B|$ Byzantine nodes, there are at most $|B|$ components that contain them. Even if these Byzantine nodes claim to be connected with each other, the size of the component is less than $n - |B|$. Otherwise, since we have an additional $|B|+1$ components, the number of vertices would be $\geq n - |B| + |B| + 1 > n$.

We know that the fragment ids of each honest node is known to everyone else after each run of Algorithm 7. At the end of the algorithm, the leader knows the fragment ids of every non-Byzantine node in the network. From the previous analysis we know that no component has size at least $n - |B|$, and hence, with high probability, the leader will reject this graph. ◀

Efficient Classification of Locally Checkable Problems in Regular Trees

Alkida Balliu ✉
Gran Sasso Science Institute, L'Aquila, Italy

Sebastian Brandt ✉
CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

Yi-Jun Chang ✉
National University of Singapore, Singapore

Dennis Olivetti ✉
Gran Sasso Science Institute, L'Aquila, Italy

Jan Studený ✉
Aalto University, Espoo, Finland

Jukka Suomela ✉
Aalto University, Espoo, Finland

Abstract

We give practical, efficient algorithms that automatically determine the asymptotic distributed round complexity of a given locally checkable graph problem in the $[\Theta(\log n), \Theta(n)]$ region, in two settings. We present one algorithm for unrooted regular trees and another algorithm for rooted regular trees. The algorithms take the description of a locally checkable labeling problem as input, and the running time is polynomial in the size of the problem description. The algorithms decide if the problem is solvable in $O(\log n)$ rounds. If not, it is known that the complexity has to be $\Theta(n^{1/k})$ for some $k = 1, 2, \ldots$, and in this case the algorithms also output the right value of the exponent k.

In rooted trees in the $O(\log n)$ case we can then further determine the exact complexity class by using algorithms from prior work; for unrooted trees the more fine-grained classification in the $O(\log n)$ region remains an open question.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases locally checkable labeling, locality, distributed computational complexity

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.8

Related Version *Full Version*: https://arxiv.org/abs/2202.08544

1 Introduction

We give practical, efficient algorithms that automatically determine the asymptotic distributed round complexity of a given *locally checkable* graph problem in *rooted or unrooted regular trees* in the $[\Theta(\log n), \Theta(n)]$ region, for both LOCAL and CONGEST models, see Section 3 for the precise definitions. In these cases, the distributed round complexity of any locally checkable problem is known to fall in one of the classes shown in Figure 1 [22, 21, 11, 20, 14, 31, 12]. Our algorithms can distinguish between all higher complexity classes from $\Theta(\log n)$ to $\Theta(n)$.

1.1 State of the art

Since 2016, there has been a large body of work studying the possible complexities of LCL problems. After an impressive sequence of works, the complexity landscape of LCL problems on bounded-degree general graphs, trees, and paths is now well-understood. For example,

© Alkida Balliu, Sebastian Brandt, Yi-Jun Chang, Dennis Olivetti, Jan Studený, and Jukka Suomela; licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 8; pp. 8:1–8:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

(a) Rooted regular trees in deterministic and randomized CONGEST and LOCAL:

$\Theta(1)$ — prior work (EXP) — $\Theta(\log^* n)$ — prior work (EXP) — $\Theta(\log n)$ — prior work (P) — $\dots$ — this work (P) — $\Theta(n^{1/3})$ — this work (P) — $\Theta(n^{1/2})$ — this work (P) — $\Theta(n)$

(b) Unrooted regular trees in deterministic CONGEST and LOCAL:

$\Theta(1)$ — unknown — $\Theta(\log^* n)$ — unknown — $\Theta(\log n)$ — this work (P) — $\dots$ — this work (P) — $\Theta(n^{1/3})$ — this work (P) — $\Theta(n^{1/2})$ — this work (P) — $\Theta(n)$

(c) Unrooted regular trees in randomized CONGEST and LOCAL:

$\Theta(1)$ — unknown — $\Theta(\log^* n)$ — unknown — $\Theta(\log \log n)$ — unknown — $\Theta(\log n)$ — this work (P) — $\dots$ — this work (P) — $\Theta(n^{1/3})$ — this work (P) — $\Theta(n^{1/2})$ — this work (P) — $\Theta(n)$

Figure 1 The most efficient algorithms for the classification of distributed round complexities. In the figure we show all possible complexity classes. Each gap between two classes corresponds to a natural decision problem: given a locally checkable problem, determine on which side of the gap its complexity is. For each gap we indicate whether a practical algorithm was provided already by prior work [9], whether it is first presented in this work, or whether the existence of such a routine is still an open question. The figure also indicates whether the algorithms are in P (polynomial time in the size of the problem description) or in EXP (exponential time in the size of the problem description).

it is known that there are no LCLs with deterministic complexity between $\omega(\log^* n)$ and $o(\log n)$. The proofs of some of the complexity gaps implies that the design of asymptotically optimal distributed algorithms can be *automated* in certain settings, leading to a series of research studying the computational complexity of automated design of asymptotically optimal distributed algorithms. See Section 2 for more details.

The most recent paper [9] in this line of research presented an algorithm that takes as input the description of an LCL problem defined in *rooted regular trees* and classifies the problem into one of the four complexity classes $O(1)$, $\Theta(\log^* n)$, $\Theta(\log n)$, and $n^{\Theta(1)}$. The classification applies to both the LOCAL and CONGEST models of distributed computing, both for randomized and deterministic algorithms.

To illustrate the setting of locally checkable problems in rooted regular trees, consider, for example, the following problem, which is meaningful for rooted binary trees:

> Each node is labeled with 1 or 2. If the label of an internal node is 1, exactly one of its two children must have label 1, and if the label of an internal node is 2, both of its children must have label 1.

We can represent it in a concise manner as a problem $\mathcal{C} = \{1 : 12, 2 : 11\}$, where $a : bc$ indicates that a node of label a can have its two children labeled with b and c, in some order. We can take such a description, feed it to the algorithm from [9], and it will output that this problem requires $\Theta(\log n)$ rounds in order to be solved in a rooted tree with n nodes.

1.2 What was missing

What the prior algorithm from [9] can do is classifying a given problem into one of the four main complexity classes $O(1)$, $\Theta(\log^* n)$, $\Theta(\log n)$, and $n^{\Theta(1)}$. However, if the complexity is $n^{\Theta(1)}$, we do not learn whether its complexity is, say, $\Theta(n)$ or $\Theta(\sqrt{n})$ or maybe $\Theta(n^{1/10})$.

There are locally checkable problems of complexity $\Theta(n^{1/k})$ for every $k = 1, 2, \ldots,$ and there have not been any *practical* algorithm that would determine the value of the exponent k for any given problem.

Furthermore, the algorithm from [9] is only applicable in rooted regular trees, while the case of unrooted trees is perhaps even more interesting.

It has been known that the problem of distinguishing between e.g. $\Theta(n)$ and $\Theta(\sqrt{n})$ is *in principle* decidable, due to the algorithm of [20]. This algorithm is, however, best seen as a theoretical construction. To the best of our knowledge, nobody has implemented it, there are no plans of implementing it, and it seems unlikely that one could classify any nontrivial problem with it using any real-world computer, due to its doubly exponential time complexity. This is the missing piece that we provide in this work.

1.3 Contributions and motivations

We present polynomial-time algorithms that determine not only whether the round complexity of a given LCL problem is $\Theta(n^{1/k})$ for some k, but they also determine the exact value of k. We give one algorithm for the case of unrooted trees and one algorithm for the case of rooted trees.

Our algorithms not only determine the asymptotic round complexity, but they also output a description of a distributed algorithm attaining this complexity. If the given LCL problem Π has optimal complexity $\Theta(n^{1/k})$, then our algorithms will output a description of a deterministic distributed algorithm that solves Π in $O(n^{1/k})$ rounds in the CONGEST model. Similarly, if the given LCL problem Π has optimal complexity $O(\log n)$, then our algorithms will output a description of a deterministic distributed algorithm that solves Π in $O(\log n)$ rounds in the CONGEST model.

We have implemented both algorithms for the case of 3-regular trees, the proof-of-concept implementations are freely available online,[1] and they work fast also in practice.

From a practical point of view, together with prior work from [9], there is now a practical algorithm that is able to *completely determine* the complexity of any LCL problem in rooted regular trees.[2] In the case of unrooted regular trees deciding between the lower complexity classes below $o(\log n)$ remains an open question.

From a theoretical point of view, this work *significantly expands* the class of LCL problems whose optimal complexity is known to be decidable in polynomial time. See Figure 1 for a summary of the current state of the art on the classification of LCL complexities for regular trees, showing where the new algorithms are applicable and where the state of the art is given by existing results.

We note that the problem of determining the optimal complexity of an LCL problem is computationally hard in general: It is undecidable in general [37], EXPTIME-hard even for bounded-degree trees [20], and PSPACE-hard even for paths and cycles with input labels [2]. Hence, in order to understand whether polynomial-time algorithms are even possible, we must restrict our consideration to restricted cases, such as LCLs with no inputs defined on regular trees. In fact, it is known that it is possible to use LCLs with no inputs defined on non-regular trees to encode LCLs with inputs, and hence, by allowing inputs, or constraints that depend on the degree of the nodes, we would make decidability at least PSPACE-hard.

[1] `https://github.com/jendas1/poly-classifier`

[2] Even though some algorithms in [9] are exponential in the size of the description of the problem, they are nevertheless very efficient in practice. In fact, the authors of [9] have implemented them for the case of binary rooted trees and they are indeed very fast in practice [41].

Motivations. Studying LCLs is interesting because, on the one hand, this class of problems is large enough to contain a significant fraction of problems that are commonly studied in the context of the LOCAL model (e.g., $(\Delta+1)$-coloring, $(2\Delta-1)$-edge coloring, Δ-coloring, weak 2-coloring, maximal matching, maximal independent set, sinkless orientation, many other orientation problems, edge splitting problems, locally maximal cut, defective colorings, ...), but, on the other hand, it is restricted enough so that we can prove interesting results about them, such as decidability and complexity gaps. Moreover, techniques used to prove results on LCLs have been already shown to be extremely useful outside the LCL context: for example, all recent results about lower bounds for locally checkable problems in the unbounded degree case – e.g., for MIS, maximal matching, ruling sets, and other fundamental problems – use techniques that originally were introduced in the context of LCLs [5, 19, 8, 6, 7].

In this work, we restrict our attention to the case of regular trees. The study of LCLs on trees is related with our understanding of graph problems in the general setting. Actually, for many problems of interest, unrooted regular trees are hard instances, and hence understanding the complexity of LCLs on trees could help us in understanding the complexity of problems in general unbounded-degree graphs. In fact, a relatively new and promising technique called *round elimination* has been used to prove tight lower bounds for interesting graph problems such as maximal matchings, maximal independent sets, and ruling sets, even if, for now, we are only able to apply this technique for proving lower bounds on trees [15, 38, 5, 8, 4, 19, 6, 7].

As for the more restrictive setting of *regular* trees, we would like to point out that many natural LCL problems have the same optimal complexity in both bounded-degree trees and regular trees. This includes, for example, the k-coloring problem. For any tree T whose maximum degree is at most Δ, we may consider the Δ-regular tree T^* which is the result of appending degree-1 nodes to all nodes v in T with $1 < \deg(v) < \Delta$ to increase the degree of v to Δ. We may locally simulate T^* in the network T. As any proper k-coloring of T^* restricting to T is also a proper k-coloring, this reduces the k-coloring problem on bounded-degree trees to the same problem on regular trees, showing that the k-coloring problem has the same optimal complexity in both graph classes. More generally, if an LCL problem Π has the property that removing degree-1 nodes preserves the correctness of a solution, then Π has the same optimal complexity in both bounded-degree trees and regular trees, so our results in this work also apply to these LCLs on bounded-degree trees.

2 Related work

Locally Checkable Labeling problems have been introduced by Naor and Stockmeyer [37], but the class of locally checkable problems has been studied in the distributed setting even before (e.g., in the context of self-stabilisation [1]). For many locally checkable problems, researchers have been trying to understand the exact time complexity, and while in many cases upper bounds have been known since the 80s, matching lower bound have been discovered only recently. Examples of this line of research relate to the problems of colorings, matchings, and independent sets, see e.g. [25, 32, 33, 39, 28, 26, 5, 40, 34, 7, 29].

In parallel, there have been many works that tried to understand these problems from a *complexity theory* point of view, trying to develop general techniques for classifying problems, understanding which complexities can actually exist, and developing generic algorithmic techniques to solve whole classes of problems at once. In particular, a broad class[3] of locally checkable problems, called *Locally Checkable Labelings* (LCLs), has been studied in the LOCAL model of distributed computing, which will be formally defined later.

[3] For example, our definition of LCL does not allow an infinite number of labels, so it does not capture some locally checkable problems such as fractional matching.

Paths and cycles. The first graph topologies on which promising results have been proved are paths and cycles. In these graphs, we now know that there are problems with the following *three* possible time complexities:

- $O(1)$: this class contains, among others, trivial problems, e.g. problems that require every node to output the same label.
- $\Theta(\log^* n)$: this class contains, for example, the 3-coloring problem [25, 32, 36].
- $\Theta(n)$: this class contains hard problems, for example the problem of consistently orient the edges of a cycle, or the 2-coloring problem.

For LCLs in paths and cycles, we know that there are no other possible complexities, that is, there are *gaps* between the above classes. In other words, there are no LCLs with a time complexity that lies between $\omega(1)$ and $o(\log^* n)$ [37], and no LCLs with a time complexity that lies between $\omega(\log^* n)$ and $o(n)$ [21]. These results hold also for *randomized* algorithms, and they are constructive: if for example we find a way to design an $O(\log n)$-rounds randomized algorithm for a problem, then we can automatically convert it into an $O(\log^* n)$-round deterministic algorithm.

Moreover, in paths and cycles, given an LCL problem, we can *decide* its time complexity. In particular, it turns out that for problems with no inputs defined on directed cycles, deciding the complexity of an LCL is as easy as drawing a diagram and staring at it for few seconds [18]. This result has later been extended to undirected cycles with no inputs [23]. Unfortunately, as soon as we consider LCLs where the constraints of the problem may depend on the given inputs, decidability becomes much harder, and it is now known to be PSPACE-hard [2], even for paths and cycles.

Trees. Another class of graphs that has been studied quite a lot is the one containing *trees*. While there are still problems with complexities $O(1)$, $\Theta(\log^* n)$, and $\Theta(n)$, there are also additional complexity classes, and sometimes here randomness can help. For example, there are problems that require $\Theta(\log n)$ rounds for both deterministic and randomized algorithms, while there are problems, like *sinkless orientation*, that require $\Theta(\log n)$ rounds for deterministic algorithms and $\Theta(\log \log n)$ rounds for randomized ones [17, 21, 30]. Moreover, there are problems with complexity $\Theta(n^{1/k})$, for any natural number $k \geq 1$ [22]. It is known that these are the only possible time complexities in trees [22, 21, 11, 20, 14, 31]. In [12], it has been shown that the same results hold also in a more restrictive model of distributed computing, called CONGEST model, and that for any given problem, its complexities in the LOCAL and in the CONGEST model, on trees, are actually the same.

Concerning decidability, the picture is not as clear as in the case of paths and cycles. As discussed in the introduction, it is decidable, *in theory*, if a problem requires $n^{\Omega(1)}$ rounds, and, in that case, it is also decidable to determine the exact exponent [22, 20], but the algorithm is very far from being practical, and in this work we address exactly this issue. Moreover, for *lower* complexities, the problem is still open. Different works tried to tackle this issue by considering restricted cases. In [4], authors showed that it is indeed possible to achieve decidability in some cases, that is, when problems are restricted to the case of unrooted regular trees, where leaves are unconstrained, and the problem uses only *two* labels. Then, promising results have been achieved in [9], where it has been shown that, if we consider *rooted* trees, then we can decide the complexity of LCLs even for $n^{o(1)}$ complexities. Unfortunately, it is very unclear if such techniques can be used to solve the problem in the general case. In fact, we still do not know if it is decidable whether a problem can be solved in $O(1)$ rounds or it requires $\Omega(\log^* n)$ rounds, and it is not known if it is decidable whether a problem can be solved in $O(\log^* n)$ rounds or it requires $\Omega(\log n)$ for

deterministic algorithms and $\Omega(\log\log n)$ for randomized ones. These two questions are very important, and understanding them may also help in understanding problems that are not restricted to regular trees of bounded degree. This is because, as already mentioned before, for many problems it happens that unrooted regular trees are hard instances, and studying the complexity of problems in these instances may give insights for understanding problems in the general setting.

General graphs. In general graphs, many more LCL complexities are possible. For example, there is a gap similar to the one between $\omega(1)$ and $o(\log^* n)$ of trees, but now it holds only up to $o(\log\log^* n)$, and we know that there are problems in the region between $\Omega(\log\log^* n)$ and $o(\log^* n)$. In fact, for any rational $\alpha \geq 1$, it is possible to construct problems with complexity $\Theta(\log^\alpha \log^* n)$ [13]. A similar statement holds for complexities between $\Omega(\log n)$ and $O(n)$ [13, 11].

There are still complexity regions in which we do not know if there are problems or not. For example, while it is known that any problem that has randomized complexity $o(\log n)$ can be sped up to $O(T_{\mathrm{LLL}})$ [22], where T_{LLL} is the distributed complexity of the constructive version of the Lovász LOCAL Lemma, the exact value of T_{LLL} is unknown, and we only known that it lies between $\Omega(\log\log n)$ and $O(\operatorname{poly}\log\log n)$ [17, 24, 27, 40]. Another problem that falls in this region is the Δ-coloring problem, for which we still do not know the exact complexity.

Another open question regards the role of randomness. In general graphs, we know that randomness can also help outside the $O(\log n)$ region [10], but we still do not know exactly when it can help and how much.

In general graphs, unfortunately, determining the complexity of a given LCL problem is undecidable. In fact, we know that this question is undecidable even on grids [37].

3 Preliminaries

Graphs. Let $G = (V, E)$ be a graph. We denote with $n = |V|$ the number of nodes of G, with Δ the maximum degree of G, and with $\deg(v)$, for $v \in V$, the degree of v. If G is a directed graph, we denote with $\deg_{\mathsf{in}}(v)$ and $\deg_{\mathsf{out}}(v)$, the indegree and the outdegree of v, respectively. The radius-r neighborhood of a node v is defined to be the subgraph of G induced by the nodes at distance at most r from v.

Model of computing. In the LOCAL model of distributed computing, the network is represented with a graph $G = (V, E)$, where the nodes correspond to computational entities, and the edges correspond to communication links. In this model, the computational power of the nodes is unrestricted, and nodes can send arbitrarily large messages to each other.

This model is synchronous, and computation proceeds in rounds. Nodes all start the computation at the same time, and at the beginning they know n (the total number of nodes), Δ (the maximum degree of the graph), and a unique ID in $\{1, \ldots, n^c\}$, for some constant $c \geq 1$, assigned to them. Then, the computation proceeds in rounds, and at each round nodes can send (possibly different) messages to each neighbor, receive messages, and perform some LOCAL computation.

At the end of the computation, each node must produce its own part of the solution. For example, in the case of the $(\Delta + 1)$-coloring problem, each node must output its own color, that must be different from the ones of its neighbors. The time complexity is measured as the worst case number of rounds required to terminate, and it is typically expressed as a function of n, Δ, and c.

4 Technical overview

Our new results build on several techniques developed in previous works [9, 23] designing polynomial-time algorithms that determine the distributed complexity of LCL problems. In this section, we first give a brief overview of these techniques, then we discuss how in this paper we build upon them and obtain our new results. The aim of this section is to present the intuition behind the results. To keep the discussion at a high level, the presentation here will be a bit imprecise.

4.1 The high-level framework

Existing algorithms for deciding the complexity of a given LCL problem are often based on the following approach.

1. Define some combinatorial property P of LCL problems.
2. Show that computing $P(\Pi)$ for a given problem Π can be done efficiently.
3. Show that Π is in a certain complexity class if and only if $P(\Pi)$ holds.

As discussed in [18, 23], any LCL Π on directed paths can be viewed as a regular language. Taking the corresponding non-deterministic automaton, we obtain a directed graph $G(\Pi)$ that represents Π on directed paths.

For example, the maximal independent set problem can be described as the automaton with states $V = \{00, 01, 10\}$ and transitions $E = \{00 \to 01, 01 \to 10, 10 \to 00, 10 \to 01\}$. Each state corresponds to a possible labeling of the two endpoints u and v of a directed edge $u \to v$. Each transition describes a valid configuration of two neighboring directed edges $u \to v$ and $v \to w$.

It has been shown [9, 18, 16, 23] that in several cases the distributed complexity of an LCL can be characterized by simple graph properties of $G(\Pi)$, even if the underlying graph class is much more complicated than directed paths. The precise definition of $G(\Pi)$ will depend on the choice of the LCL formalism.

4.2 Paths and cycles

It was shown in [18, 23] that the distributed complexity and solvability of Π on paths and cycles can be characterized by simple graph properties of $G(\Pi)$. In particular, Π on directed cycles is solvable in $O(\log^* n)$ rounds if and only if $G(\Pi)$ contains a node v that is *path-flexible*, in the sense that there exists a number K such that, in $G(\Pi)$, there is a length-k returning walk for v, for each $k \geq K$. If such a path-flexible node v exists in $G(\Pi)$, then Π on directed cycles can be solved in $O(\log^* n)$ rounds in the following manner.

1. In $O(\log^* n)$ rounds, compute an independent set I such that the distance between the nodes in I is at least K and at most $2K$.
2. Fix the labels for the nodes in I according to the path-flexible node v in $G(\Pi)$.
3. By the path-flexibility of v, this partial labeling can be completed into a correct complete labeling.

For example, in the automaton for maximal independent set described above, the state 01 is flexible, as for each $k \geq 5$, there is a length-k walk starting and ending at 01, so a maximal independent set can be found in $O(\log^* n)$ rounds on directed cycles via the above algorithm.

The above characterization can be generalized to both paths and cycles, undirected and directed, after some minor modifications, see [23] for the details. For further examples of representing LCLs as automata and how the round complexity of an LCL can be inferred from basic properties of its associated automaton, see [18, Fig. 3] and [23, Fig. 1 and 3].

4.3 The $O(\log n)$ complexity class in regular trees

Subsequently, it was shown in [9, 16] that the class of $O(\log n)$-round solvable LCL problems on rooted and unrooted regular trees can be characterized in a similar way, based on the notion of path-flexibility in the directed graph $G(\Pi)$. To keep the discussion at a high level, we do not discuss the difference between rooted and unrooted trees here. Roughly speaking, Π can be solved in $O(\log n)$ rounds on rooted or unrooted regular trees if and only if there exists a subset of labels S such that, if we restrict Π to S, then its corresponding directed graph is strongly connected and contains a path-flexible node. Such a set S of labels is also called a *certificate* for $O(\log n)$-round solvability.[4]

A key property of such a directed graph is that there exists a number K such that, for each pair of nodes (u, v), and for each integer $k \geq K$, there is a length-k walk from u to v (here we allow the possibility of $u = v$). The property can be described in the following more intuitive manner. For any path of length at least K, regardless of how we fix the labels of its two endpoints using S, it is always possible to complete the partial labeling into a correct labeling w.r.t. Π of the entire path using only labels in S.

The intuition behind such a characterization is the fact [22] that all LCLs solvable in $O(\log n)$ rounds on bounded-degree trees can be solved in a canonical way based on *rake-and-compress decompositions.* Roughly speaking, a rake-and-compress process is a procedure that decomposes a tree by iteratively removing degree-1 nodes (rake) and removing degree-2 nodes (compress). This process partitions the set of nodes into several parts: $V = V_1^{\mathsf{R}} \cup V_1^{\mathsf{C}} \cup V_2^{\mathsf{R}} \cup V_2^{\mathsf{C}} \cup \cdots \cup V_L^{\mathsf{R}}$, where V_i^{R} is the set of nodes removed by the rake operation in the ith iteration and V_i^{C} is the set of nodes removed by the compress operation in the ith iteration. It can be shown that $L = O(\log n)$ [35].

There are several variants of a rake-and-compress process. Here the considered variant is such that, in the compress operation, a degree-2 node v is removed if v belongs to a path whose length is at least ℓ, so we may assume that the connected components in the subgraph induced by V_i^{C} are paths with length at least ℓ.

Let Π be any LCL problem satisfying the combinatorial characterization for $O(\log n)$-round solvability discussed above, and let the set of labels S be a certificate for $O(\log n)$-round solvability. By setting $\ell = K$ in the property of the combinatorial characterization, we may obtain an $O(\log n)$-round algorithm solving the given LCL problem Π using only the labels in S. The high-level idea is that we can label the tree in an order that is the reverse of the one of the rake-and-compress procedure: $V_L^{\mathsf{R}}, \ldots, V_2^{\mathsf{C}}, V_2^{\mathsf{R}}, V_1^{\mathsf{C}}, V_1^{\mathsf{R}}$, as we observe that the property of the combinatorial characterization discussed above ensures that any correct labeling of $V_L^{\mathsf{R}} \cup \cdots \cup V_i^{\mathsf{R}}$ can be extended to a correct labeling of $V_L^{\mathsf{R}} \cup \cdots \cup V_i^{\mathsf{R}} \cup V_{i-1}^{\mathsf{C}}$ and similarly any correct labeling of $V_L^{\mathsf{R}} \cup \cdots \cup V_i^{\mathsf{C}}$ can be extended to a correct labeling of $V_L^{\mathsf{R}} \cup \cdots \cup V_i^{\mathsf{C}} \cup V_i^{\mathsf{R}}$.

The requirement that Π is an LCL problem defined on *regular* trees is *critical* in the above approach, as this requirement ensures that for each non-leaf node, the set of constraints is the same, so we do not need to worry about the possibility for different nodes in the tree to have different sets of constraints in Π. Indeed, if we allow nodes of different degrees to have different sets of constraints, then the problem of determining the distributed complexity of an LCL in bounded-degree trees becomes EXPTIME-hard [20].

[4] Although the certificate described in [9] also includes the steps in the construction of S, the set S alone suffices to certify that Π can be solved in $O(\log n)$ rounds, as the $O(\log n)$-round algorithm described in [9] uses only S.

4.4 The polynomial complexity region in regular trees

In this work, we will extend the above approach to cover all complexity classes in the $[\Theta(\log n), \Theta(n)]$ region. By [11, 20, 22], we know that the possible complexity classes in this region are $\Theta(\log n)$ and $\Theta(n^{1/k})$ for all positive integers k. Similar to the complexity class $O(\log n)$, any LCL problem Π solvable in $O(n^{1/k})$ rounds can be solved in a canonical way in $O(n^{1/k})$ rounds using a variant of rake-and-compress decomposition [20].

Specifically, Π is $O(n^{1/k})$-round solvable if and only if it can be solved in a canonical way using a rake-and-compress decomposition, where in each iteration, we perform $\gamma = O(n^{1/k})$ rake operations and one compress operation. Similar to the case of complexity class $O(\log n)$, in the compress operation, a degree-2 node v is removed if v belongs to a path whose length is at least ℓ, where $\ell = O(1)$ is some sufficiently large number depending only on the LCL problem Π. It can be shown [20] that by selecting $\gamma = O(n^{1/k})$ to be large enough, the number of layers L in the decomposition $V = V_1^{\mathsf{R}} \cup V_1^{\mathsf{C}} \cup V_2^{\mathsf{R}} \cup V_2^{\mathsf{C}} \cup \cdots \cup V_L^{\mathsf{R}}$ is k, and such a decomposition can be computed in $O(n^{1/k})$ rounds.

To derive a certificate for $O(n^{1/k})$-round solvability based on the result of [20], we will need to take into consideration the following properties about the variant of the rake-and-compress decomposition described above.

- The number of layers $L = k$ is now a *finite* number independent of the size of the graph n. For technical reasons, this means that the certificate for $O(n^{1/k})$-round solvability cannot be based on a single set of labels S, as the certificate for $O(\log n)$-round solvability [9, 16]. We need to consider the possibility that different sets of labels are used for different layers in the design of the certificate for $O(n^{1/k})$-round solvability.
- The number of rake operations for a layer can be unbounded as n goes to infinity. That is, V_i^{R} is no longer an independent set, and each connected component in the subgraph induced by V_i^{R} can be a very large tree.

The certificate. Our certificate for $O(n^{1/k})$-round solvability will be based on the notion of a *good* sequence of sets of labels. The definition of a good sequence relies on two functions on a set of labels: trim and flexible-SCC. As we will later see, these two functions correspond to rake and compress, respectively. Given an LCL problem Π and a set of labels S, trim(S) and flexible-SCC are defined as follows.

- trim(S) is the subset of S resulting from removing all labels $\sigma \in S$ meeting the following conditions: There exists some number i such that if the root of the complete regular tree T of height i is labeled by σ, then we are not able to complete the labeling of T using only labels in S such that the overall labeling is correct w.r.t. Π.
- flexible-SCC(S) is a collection of disjoint subsets of S defined as follows. Consider the directed graph representing the LCL problem Π restricted to S. Let flexible-SCC(S) be the set of strongly connected components that have a path-flexible node. The intuition behind this definition is similar to the intuition behind the certificate for $O(\log n)$-round solvability.

We briefly explain the connection between trim and rake. Suppose we want to find a correct labeling of a regular tree T using only the labels in S. If a label σ is in trim(S), then σ can only be used in places that are sufficiently close to a leaf. To put it another way, if we do a large number of rakes to T, then the labels in trim(S) can only be used to label the nodes that removed due to a rake operation.

The connection between $\mathsf{flexible\text{-}SCC}$ to compress is due to the fact that the nodes removed due to a compress operation form long paths, and we know that in order to label long paths efficiently in $O(\log^* n)$ rounds, it is necessary to use labels corresponding to path-flexible nodes, due to the existing automata-theoretic characterization [18, 23] of round complexity of LCLs on paths and cycles.

We say that a sequence $(\Sigma_1^{\mathsf{R}}, \Sigma_1^{\mathsf{C}}, \Sigma_2^{\mathsf{R}}, \Sigma_2^{\mathsf{C}}, \ldots, \Sigma_k^{\mathsf{R}})$ is *good* if it satisfies the following rules, where Σ is the set of all labels of Π.

$$\Sigma_i^{\mathsf{R}} = \begin{cases} \mathsf{trim}(\Sigma) & \text{if } i = 1, \\ \mathsf{trim}(\Sigma_{i-1}^{\mathsf{C}}) & \text{if } i > 1. \end{cases}$$

$$\Sigma_i^{\mathsf{C}} \in \mathsf{flexible\text{-}SCC}(\Sigma_i^{\mathsf{R}}).$$

$$\Sigma_k^{\mathsf{R}} \neq \emptyset.$$

The only nondeterminism in the above rules is the choice of $\Sigma_i^{\mathsf{C}} \in \mathsf{flexible\text{-}SCC}(\Sigma_i^{\mathsf{R}})$ for each i. We will show that such a sequence exists if and only if the underlying LCL problem can be solved in $O(n^{1/k})$ rounds. Intuitively, Σ_i^{R} represents the set of labels that are eligible to label the nodes in V_i^{R}, and similarly Σ_i^{C} represents the set of labels that are eligible to label the nodes in V_i^{C}.

The classification. The notion of a good sequence allows us to classify the complexity classes in the region $[\Theta(\log n), \Theta(n)]$. Specifically, we define the *depth* d_Π of an LCL problem Π as the largest k such that a good sequence $(\Sigma_1^{\mathsf{R}}, \Sigma_1^{\mathsf{C}}, \Sigma_2^{\mathsf{R}}, \Sigma_2^{\mathsf{C}}, \ldots, \Sigma_k^{\mathsf{R}})$ exists. If there is no good sequence, then we set $d_\Pi = 0$. If there is a good sequence $(\Sigma_1^{\mathsf{R}}, \Sigma_1^{\mathsf{C}}, \Sigma_2^{\mathsf{R}}, \Sigma_2^{\mathsf{C}}, \ldots, \Sigma_k^{\mathsf{R}})$ for each positive integer k, then we set $d_\Pi = \infty$. We will show that d_Π characterizes the distributed complexity of Π in the following manner.

- If $d_\Pi = 0$, then Π is unsolvable in the sense that there exists a regular tree such that there is no correct solution of Π on this rooted tree. This follows from the definition of trim and the observation that $d_\Pi = 0$ if $\mathsf{trim}(\Sigma) = \emptyset$.
- If $d_\Pi = k$ is a positive integer, then the distributed complexity of Π is $\Theta(n^{1/k})$.
- If $d_\Pi = \infty$, then Π can be solved in $O(\log n)$ rounds. If we can have a good sequence that is arbitrarily long, then there must be a *fixed point* S in the sequence such that $\mathsf{trim}(S) = S$ and $\mathsf{flexible\text{-}SCC}(S) = \{S\}$, because $\Sigma_1^{\mathsf{R}} \supseteq \Sigma_1^{\mathsf{C}} \supseteq \cdots \supseteq \Sigma_k^{\mathsf{R}}$. We will show that the fixed point S qualifies to be a certificate for $O(\log n)$-round solvability.

The fixed point phenomenon explains why the notion of good sequence was not needed in [9, 16], as the existence of a fixed point for the case Π is $O(\log n)$-round solvable implies that we may apply the same strategy according to the fixed point to label each layer of the rake-and-compress decomposition to solve Π in $O(\log n)$ rounds.

To show correctness and efficiency of our characterization, we do the following.

Upper bound: Given a good sequence $(\Sigma_1^{\mathsf{R}}, \Sigma_1^{\mathsf{C}}, \Sigma_2^{\mathsf{R}}, \Sigma_2^{\mathsf{C}}, \ldots, \Sigma_k^{\mathsf{R}})$, show that there exists an $O(n^{1/k})$-round algorithm solving Π. Therefore, $d_\Pi = k$ implies $O(n^{1/k})$-round solvability.

Lower bound: Given an $o(n^{1/k})$-round algorithm solving Π, show that a good sequence $(\Sigma_1^{\mathsf{R}}, \Sigma_1^{\mathsf{C}}, \Sigma_2^{\mathsf{R}}, \Sigma_2^{\mathsf{C}}, \ldots, \Sigma_{k+1}^{\mathsf{R}})$ exists. Therefore, $d_\Pi = k$ implies $\Omega(n^{1/k})$-round solvability.

Efficiency: Design a polynomial-time algorithm that computes d_Π for any given description of an LCL problem Π.

The upper bound proof is relatively simple. Similar to the certificate $O(\log n)$-round solvability, we just need to show that Π can be solved in $O(n^{1/k})$ rounds using rake-and-compress decompositions given that a good sequence $(\Sigma_1^{\mathsf{R}}, \Sigma_1^{\mathsf{C}}, \Sigma_2^{\mathsf{R}}, \Sigma_2^{\mathsf{C}}, \ldots, \Sigma_k^{\mathsf{R}})$ exists.

The lower bound proof is much more complicated. Given an algorithm $\mathcal{A}$ solving Π in $t = o(n^{1/k})$ rounds, we will consider a tree G that is a result of a hierarchical combination of complete trees and paths of length greater than t. Intuitively, G is chosen to be the fullest possible tree that can be partitioned into $V = V_1^{\mathsf{R}} \cup V_1^{\mathsf{C}} \cup V_2^{\mathsf{R}} \cup V_2^{\mathsf{C}} \cup \cdots \cup V_{k+1}^{\mathsf{R}}$ with a rake-and-compress decomposition of [20] with $L = k+1$ layers. We will prove by induction that if we take Σ_i^{R} to be the set of possible output labels of $\mathcal{A}$ for $V_i^{\mathsf{R}} \cup V_i^{\mathsf{C}} \cup \cdots \cup V_{k+1}^{\mathsf{R}}$ and take Σ_i^{C} to be the set of possible output labels of $\mathcal{A}$ for $V_i^{\mathsf{C}} \cup V_{i+1}^{\mathsf{R}} \cup \cdots \cup V_{k+1}^{\mathsf{R}}$, then $(\Sigma_1^{\mathsf{R}}, \Sigma_1^{\mathsf{C}}, \Sigma_2^{\mathsf{R}}, \Sigma_2^{\mathsf{C}}, \ldots, \Sigma_{k+1}^{\mathsf{R}})$ must be a good sequence. In particular, the non-emptiness of $\Sigma_{k+1}^{\mathsf{R}}$ follows from the correctness of $\mathcal{A}$.

To design a polynomial-time algorithm computing d_Π, we recall that the only nondeterminism in the rules for a good sequence is the choice of $\Sigma_i^{\mathsf{C}} \in \textsf{flexible-SCC}(\Sigma_i^{\mathsf{R}})$, so we will just do a brute-force search for all possibilities. Although this seems inefficient, we recall that $\textsf{flexible-SCC}(\Sigma_i^{\mathsf{R}})$ is a collection of disjoint subsets of Σ_i^{R}, so the sum of the size of all sets of labels considered in each level is at most the total number of labels $|\Sigma|$ in Π. The number of levels we need to explore is also bounded, as $\Sigma_1^{\mathsf{R}} \supseteq \Sigma_1^{\mathsf{C}} \supseteq \cdots \supseteq \Sigma_k^{\mathsf{R}}$. If k exceeds $|\Sigma|$, then we know that there is a fixed point Σ_i^{R} such that $\Sigma_i^{\mathsf{R}} = \Sigma_i^{\mathsf{C}} = \Sigma_{i+1}^{\mathsf{R}} = \Sigma_{i+1}^{\mathsf{C}} = \cdots$, so $d_\Pi = \infty$.

The differences between rooted and unrooted trees. The high-level proof strategy presented in this technical overview applies to both rooted and unrooted regular trees, showing that these two graph classes behave very similarly in the complexity region $[\Theta(\log n), \Theta(n)]$. There are still some technical differences between rooted and unrooted trees.

- The formalisms for representing LCL problems are different for rooted and unrooted trees. In the case of rooted trees, the problem can refer to orientations. For example, what is permitted for a parent can be different from what is permitted for a child. Instead of specifying node and edge configurations, we follow [9] and specify what are permitted multisets of child labels for each node label.
- For the upper bound, we need to generalize the rake-and-compress decomposition of [20] so that it is applicable in rooted trees.
- For the lower bound, the lower bound graph for unrooted trees does not work for the rooted trees. Roughly speaking, this is because the presence of edge orientation increases the symmetry breaking capability of nodes, so some indistinguishability arguments in the lower bound proof for unrooted trees do not work for rooted trees. Therefore, we will need to consider a different approach for crafting the lower bound graph for rooted trees.

5 Unrooted trees

In this section, we give a polynomial-time-computable characterization of LCL problems for regular unrooted trees with complexity $O(\log n)$ or $\Theta(n^{1/k})$ for any positive integer k. Due to the page limit, our results in regular rooted trees are left to the full version [3] of the paper. A Δ-regular tree is a tree where the degree of each node is either 1 or Δ. An LCL problem for Δ-regular unrooted trees is defined as follows.

▶ **Definition 1** (LCL problems for regular unrooted trees)**.** *For unrooted trees, an* LCL *problem* $\Pi = (\Delta, \Sigma, \mathcal{V}, \mathcal{E})$ *is defined by the following components.*

- Δ *is a positive integer specifying the maximum degree.*
- Σ *is a finite set of labels.*
- $\mathcal{V}$ *is a set of size-*Δ *multisets of labels in* Σ *specifying the node constraint.*
- $\mathcal{E}$ *is a set of size-*2 *multisets of labels in* Σ *specifying the edge constraint.*

We call a size-Δ multiset C of labels in Σ a *node configuration*. A node configuration C is correct with respect to $\Pi = (\Delta, \Sigma, \mathcal{V}, \mathcal{E})$ if $C \in \mathcal{V}$. We call a size-2 multiset D of labels in Σ an *edge configuration*. An edge configuration D is correct with respect to $\Pi = (\Delta, \Sigma, \mathcal{V}, \mathcal{E})$ if $D \in \mathcal{E}$. We define the correctness criteria for a labeling of a Δ-regular tree in Definition 2.

▶ **Definition 2** (Correctness criteria). *Let $G = (V, E)$ be a tree whose maximum degree is at most Δ. For each edge $e = \{u, v\} \in E$, there are two half-edges (u, e) and (v, e). A solution of $\Pi = (\Delta, \Sigma, \mathcal{V}, \mathcal{E})$ on G is a labeling that assigns a label in Σ to each half-edge in G.*

- *For each node $v \in V$ with $\deg(v) = \Delta$ its node configuration C is the multiset of Δ half-edge labels of (v, e_1), (v, e_2), ..., (v, e_Δ), where $e_1, e_2, \ldots, e_\Delta$ are the Δ edges incident to v. We say that the labeling is locally-consistent on v if $C \in \mathcal{V}$.*
- *For each edge $e = \{u, v\} \in E$, its edge configuration D is the multiset of two half-edge labels of (u, e) and (v, e). We say that the labeling is locally-consistent on e if $D \in \mathcal{E}$.*

The labeling is a correct solution if it is locally-consistent on all $v \in V$ with $\deg(v) = \Delta$ and all $e \in E$.

In other words, a labeling of $G = (V, E)$ is correct if the edge configuration for each $e \in E$ is correct and the node configuration for each $v \in V$ with $\deg(v) = \Delta$ is correct. All nodes whose degree is not Δ are unconstrained.

Although $\Pi = (\Delta, \Sigma, \mathcal{V}, \mathcal{E})$ is defined for Δ-regular unrooted trees, Definition 2 applies to all trees whose maximum degree is at most Δ. We emphasize that all nodes v whose degree is not Δ are *unconstrained* in that there is no requirement about the node configuration of v. Nevertheless, we may focus on Δ-regular unrooted trees without loss of generality. The reason is that for any unrooted tree G whose maximum degree is at most Δ, we may consider the unrooted tree G^* which is the result of appending degree-1 nodes to all nodes v in G with $1 < \deg(v) < \Delta$ to increase the degree of v to Δ. This only blows up the number of nodes by at most a Δ factor. We claim that the asymptotic optimal round complexity of Π is the same in both G and G^*. Any correct solution of Π on G^* restricted to G is a correct solution of Π on G, as all nodes whose degree is not Δ are unconstrained. Therefore, if we have an algorithm for Π in Δ-regular unrooted trees, then the same algorithm also allows us to solve Π in unrooted trees with maximum degree Δ in the same asymptotic round complexity.

▶ **Definition 3** (Complete trees of height i). *We define the rooted trees T_i and T_i^* recursively as follows.*

- *T_0 is the trivial tree with only one node.*
- *T_i is the result of appending $\Delta - 1$ trees T_{i-1} to the root r.*
- *T_i^* is the result of appending Δ trees T_{i-1} to the root r.*

Observe that T_i^* is the unique maximum-size tree of maximum degree Δ and height i. All nodes within distance $i - 1$ to the root r in T_i^* have degree Δ. All nodes whose distance to r is exactly i are degree-1 nodes. Although T_i and T_i^* are defined as rooted trees, they can also be viewed as unrooted trees.

▶ **Definition 4** (Trimming). *Given an* LCL *problem $\Pi = (\Delta, \Sigma, \mathcal{V}, \mathcal{E})$ and a subset $\mathcal{S} \subseteq \mathcal{V}$ of node configurations, we define $\mathsf{trim}(\mathcal{S})$ as the set of all node configurations $C \in \mathcal{S}$ such that for each $i \geq 1$ it is possible to find a correct labeling of T_i^* such that the node configuration of the root is C and the node configurations of the remaining degree-Δ nodes are in $\mathcal{S}$.*

In the definition, note that if for some $i \geq 1$ it is not possible to find such a labeling of T_i^*, then it is also not possible for any larger i. The reason is that if such a labeling for larger i exists, then by taking subgraph, we obtain such a labeling for of T_i^*. Here we use the fact that nodes by taking subgraph, and using the fact that all nodes whose degree is not Δ are unconstrained.

Intuitively, $\mathsf{trim}(\mathcal{S})$ is the subset of $\mathcal{S}$ resulting from removing all node configurations in $\mathcal{S}$ that are not usable in a correct labeling of a sufficiently large Δ-regular tree using only node configurations in $\mathcal{S}$.

In fact, given any tree G of maximum degree Δ and a node v of degree Δ in G, after labeling the half-edges surrounding v using a node configuration in $\mathsf{trim}(\mathcal{S})$, it is always possible to extend this labeling to a complete correct labeling of G using only node configurations in $\mathsf{trim}(\mathcal{S})$. Such a labeling extension is possible due to Lemma 5.

▶ **Lemma 5** (Property of trimming). *Let $\mathcal{S} \subseteq \mathcal{V}$ such that $\mathsf{trim}(\mathcal{S}) \neq \emptyset$. For each node configuration $C \in \mathsf{trim}(\mathcal{S})$ and each label $\sigma \in C$, there exist a node configuration $C' \in \mathsf{trim}(\mathcal{S})$ and a label $\sigma' \in C'$ such that the multiset $\{\sigma, \sigma'\}$ is in $\mathcal{E}$.*

Proof. Assuming that such C' and σ' do not exist, we derive a contradiction as follows. We pick s to be the smallest number such that there is no correct labeling of T^*_s where the node configuration of the root r is in $\mathcal{S} \setminus \mathsf{trim}(\mathcal{S})$ and the node configuration of each remaining degree-Δ node of T^*_s is in $\mathcal{S}$. Such a number s exists due to the definition of trim.

Now consider a correct labeling of T^*_{s+1} where the node configuration of the root r is C and the node configuration of each remaining degree-Δ node is in $\mathcal{S}$. Such a correct labeling exists due to the fact that $C \in \mathsf{trim}(\mathcal{S})$. Our assumption on the non-existence of C' and σ' implies that the node configuration $\tilde{C}$ of one child w of the root r of T^*_{s+1} must be in $\mathcal{S} \setminus \mathsf{trim}(\mathcal{S})$. However, the radius-$s$ neighborhood of w in T^*_{s+1} is isomorphic to T^*_s rooted at w. Since the node configuration of w is in $\mathcal{S} \setminus \mathsf{trim}(\mathcal{S})$, our choice of s implies that the labeling of the radius-s neighborhood of w cannot be correct, which is a contradiction. ◀

Path-form of an LCL problem. Given an LCL problem $\Pi = (\Delta, \Sigma, \mathcal{V}, \mathcal{E})$ and a subset $\mathcal{S} \subseteq \mathcal{V}$ of node configurations, we define

$$\mathcal{D}_{\mathcal{S}} = \text{the set of all size-2 multisets } D \text{ such that } D \text{ is a sub-multiset of } C \text{ for some } C \in \mathcal{S}.$$

To understand the intuition behind the definition $\mathcal{D}_{\mathcal{S}}$, define the length-$k$ hairy path H_k as the result obtained by starting from a length-k path $P = (v_1, v_2, \ldots, v_{k+1})$ and then adding degree-1 nodes to make $\deg(v_i) = \Delta$ for all $1 \leq i \leq k+1$. If our task is to label hairy paths using node configurations in $\mathcal{S}$, then this task is identical to labeling paths using node configurations in $\mathcal{D}_{\mathcal{S}}$. In other words, the LCL problem $(\Delta, \Sigma, \mathcal{S}, \mathcal{E})$ on hairy paths is equivalent to the LCL problem $(2, \Sigma, \mathcal{D}_{\mathcal{S}}, \mathcal{E})$ on paths. Hence $(2, \Sigma, \mathcal{D}_{\mathcal{S}}, \mathcal{E})$ is the *path-form* of $(\Delta, \Sigma, \mathcal{S}, \mathcal{E})$.

Automaton for the path-form of an LCL problem. Given a set $\mathcal{D}$ of size-2 multisets whose elements are in Σ, we define the directed graph $\mathcal{M}_{\mathcal{D}}$ as follows. The node set $V(\mathcal{M}_{\mathcal{D}})$ of $\mathcal{M}_{\mathcal{D}}$ is the set of all pairs $(a, b) \in \Sigma^2$ such that the multiset $\{a, b\}$ is in $\mathcal{D}$. The edge set $E(\mathcal{M}_{\mathcal{D}})$ of $\mathcal{M}_{\mathcal{D}}$ is defined as follows. For any two pairs $(a, b) \in V(\mathcal{M}_{\mathcal{D}})$ and $(c, d) \in V(\mathcal{M}_{\mathcal{D}})$, we add a directed edge $(a, b) \to (c, d)$ if the multiset $\{b, c\}$ is an edge configuration in $\mathcal{E}$. Note that $\mathcal{M}_{\mathcal{D}}$ could contain self-loops.

The motivation for considering $\mathcal{M}_{\mathcal{D}}$ is that it can be seen as an automaton recognizing the correct solutions for the LCL problem $(2, \Sigma, \mathcal{D}, \mathcal{E})$ on paths, as each length-k walk $(a_1, b_1) \to (a_2, b_2) \to \cdots \to (a_{k+1}, b_{k+1})$ of $\mathcal{M}_{\mathcal{D}}$ corresponds to a correct labeling of a length-k path $(v_1, v_2, \ldots, v_{k+1})$ where the labeling of half-edge $(v_i, \{v_{i-1}, v_i\})$ is a_i and the labeling of half-edge $(v_i, \{v_i, v_{i+1}\})$ is b_i.

Path-flexibility. With respect to the directed graph $\mathcal{M}_{\mathcal{D}}$, we say that $(a,b) \in V(\mathcal{M}_{\mathcal{D}})$ is path-flexible if there exists an integer K such that for each integer $k \geq K$, there exist length-k walks $(a,b) \rightsquigarrow (a,b)$, $(a,b) \rightsquigarrow (b,a)$, $(b,a) \rightsquigarrow (a,b)$, and $(b,a) \rightsquigarrow (b,a)$ in $\mathcal{M}_{\mathcal{D}}$. Throughout this paper, we write $u \rightsquigarrow v$ to denote a walk starting from u and ending at v.

It is clear that (a,b) is path-flexible if and only if (b,a) is path-flexible. Hence we may extend the notion of path-flexibility from $V(\mathcal{M}_{\mathcal{D}})$ to $\mathcal{D}$. That is, we say that a size-2 multiset $\{a,b\} \in \mathcal{D}$ is path-flexible if (a,b) is path-flexible.

The following lemma is useful in lower bound proofs. For any $\{a,b\} \in \mathcal{D}$ that is not path-flexible, the following lemma shows that there are infinitely many path lengths k such that there is no length-k $s \rightsquigarrow t$ walk for some $s \in \{(a,b),(b,a)\}$ and $t \in \{(a,b),(b,a)\}$. As we will later see, this inflexibility in the possible path lengths implies lower bounds for distributed algorithms that may use the configuration $\{a,b\}$.

▶ **Lemma 6** (Property of path-inflexibility). *Suppose that the size-2 multiset $\{a,b\} \in \mathcal{D}$ is not path-flexible. Then one of the following holds.*

- *There is no $s \rightsquigarrow t$ walk for at least one choice of $s \in \{(a,b),(b,a)\}$ and $t \in \{(a,b),(b,a)\}$.*
- *There is an integer $2 \leq x \leq |\Sigma|^2$ such that for any positive integer k that is not an integer multiple of x, there are no length-k walks $(a,b) \rightsquigarrow (a,b)$ and $(b,a) \rightsquigarrow (b,a)$ in $\mathcal{M}_{\mathcal{D}}$.*

Proof. Suppose that $\{a,b\} \in \mathcal{D}$ is not path-flexible. We assume that there are $s \rightsquigarrow t$ walks for all choices of $s \in \{(a,b),(b,a)\}$ and $t \in \{(a,b),(b,a)\}$. To prove this lemma, it suffices to show that there is an integer $2 \leq x \leq |\Sigma|^2$ such that for any positive integer k that is not an integer multiple of x, there are no length-k walks $(a,b) \rightsquigarrow (a,b)$ and $(b,a) \rightsquigarrow (b,a)$.

First of all, we claim that for any integer K there is an integer $k \geq K$ such that there is no length-k walk $(a,b) \rightsquigarrow (a,b)$. If this claim does not hold, then there is an integer K such that there is a length-k walk $(a,b) \rightsquigarrow (a,b)$ for each $k \geq K$. Combining these walks with existing walks $(a,b) \rightsquigarrow (b,a)$ and $(b,a) \rightsquigarrow (a,b)$, we infer that there exists an integer K' such that for each integer $k \geq K'$, there exist length-k walks $(a,b) \rightsquigarrow (a,b)$, $(a,b) \rightsquigarrow (b,a)$, $(b,a) \rightsquigarrow (a,b)$, and $(b,a) \rightsquigarrow (b,a)$ in $\mathcal{M}_{\mathcal{D}}$, contradicting the assumption that $\{a,b\} \in \mathcal{D}$ is not path-flexible.

Let U be the set of integers k such that there is a length-k walk $(a,b) \rightsquigarrow (a,b)$. Note that by taking reversal, the existence of a length-k walk $(a,b) \rightsquigarrow (a,b)$ implies the existence of a length-k walk $(b,a) \rightsquigarrow (b,a)$, and vice versa. Our assumption on the existence of a walk $(a,b) \rightsquigarrow (a,b)$ implies $U \neq \emptyset$. We choose $x = \gcd(U)$ to be the greatest common divisor of U, so that for any integer k that is not an integer multiple of x, there are no length-k walks $(a,b) \rightsquigarrow (a,b)$ and $(b,a) \rightsquigarrow (b,a)$ in $\mathcal{M}_{\mathcal{D}}$. We must have $x \geq 2$ because there cannot be two co-prime numbers in U, since otherwise there exists an integer K such that U includes all integers that are at least K, contradicting the claim proved above. Specifically, if the two co-prime numbers are k_1 and k_2, then we may set $K = g(k_1,k_2) + 1 = k_1k_2 - k_1 - k_2 + 1$, where $g(k_1,k_2)$ is the Frobenius number of the set $\{k_1,k_2\}$ [42]. We also have $x \leq |\Sigma|^2$, since the smallest number in U is at most the number of nodes in $\mathcal{M}_{\mathcal{D}}$, which is upper bounded by $|\Sigma|^2$. ◀

For the special case of $|\Sigma| = 1$ and $\mathcal{D} \neq \emptyset$, we must have $a = b$ in Lemma 6. Since there is no integer x satisfying $2 \leq x \leq |\Sigma|^2$ when $|\Sigma| = 1$, Lemma 6 implies that if $\{a,a\}$ is not path-flexible, then there is no walk $(a,a) \rightsquigarrow (a,a)$, where $\{a,a\}$ is the unique element in $\mathcal{D}$.

Path-flexible strongly connected components. Since each $\{a,b\} \in \mathcal{D}$ corresponds to two nodes (a,b) and (b,a) in $\mathcal{M}_{\mathcal{D}}$, we will consider a different notion of a strongly connected component. In Definition 7, we do not require the elements a, b, c, and d to be distinct. For example, we may have $\{a,b\} = \{c,d\}$ or $a = b$.

▶ **Definition 7** (Strongly connected components). *Let $\mathcal{D}$ be a set of size-2 multisets of elements in Σ. For each $\{a,b\} \in \mathcal{D}$ and $\{c,d\} \in \mathcal{D}$, we write $\{a,b\} \sim \{c,d\}$ if there is a walk $s \rightsquigarrow t$ in $\mathcal{M}_{\mathcal{D}}$ for each choice of $s \in \{(a,b),(b,a)\}$ and $t \in \{(c,d),(d,c)\}$.*

Let $\mathcal{D}^{\sim}$ be the set of all $\{a,b\} \in \mathcal{D}$ such that $\{a,b\} \sim \{a,b\}$. Then we define the strongly connected components of $\mathcal{D}$ as the equivalence classes of $\sim$ over $\mathcal{D}^{\sim}$.

By taking reversal, the existence of an $(a,b) \rightsquigarrow (c,d)$ walk implies the existence of a $(d,c) \rightsquigarrow (b,a)$ walk. Therefore, if there is a walk $s \rightsquigarrow t$ in $\mathcal{M}_{\mathcal{D}}$ for each choice of $s \in \{(a,b),(b,a)\}$ and $t \in \{(c,d),(d,c)\}$, then there is also a walk $t \rightsquigarrow s$ in $\mathcal{M}_{\mathcal{D}}$ for each choice of $s \in \{(a,b),(b,a)\}$ and $t \in \{(c,d),(d,c)\}$. Hence the relation $\sim$ in Definition 7 is symmetric over $\mathcal{D}$. It is clear from the definition of $\sim$ in Definition 7 that it is transitive over $\mathcal{D}$ and it is reflexive over $\mathcal{D}^{\sim}$, so $\sim$ is indeed an equivalence relation over $\mathcal{D}^{\sim}$.

For any strongly connected component $\mathcal{D}'$ of $\mathcal{D}$, it is clear that either all $\{a,b\} \in \mathcal{D}'$ are path-flexible or all $\{a,b\} \in \mathcal{D}'$ are not path-flexible. We say that a strongly connected component $\mathcal{D}'$ is path-flexible if all $\{a,b\} \in \mathcal{D}'$ are path-flexible. We define $\mathsf{flexibility}(\mathcal{D}')$ as the minimum number K such that for each integer $k \geq K$ there is an $(a,b) \rightsquigarrow (c,d)$ walk of length k for all choices of a, b, c, and d such that $\{a,b\} \in \mathcal{D}'$ and $\{c,d\} \in \mathcal{D}'$. Such a number K exists given that $\mathcal{D}'$ is a path-flexible strongly connected component. We define

$$\mathsf{flexible\text{-}SCC}(\mathcal{D}) = \begin{array}{l}\text{the set of all subsets of } \mathcal{D} \text{ that are a path-}\\ \text{flexible strongly connected component of } \mathcal{D}.\end{array}$$

Clearly, elements in $\mathsf{flexible\text{-}SCC}(\mathcal{D})$ are disjoint subsets of $\mathcal{D}$. It is possible that $\mathsf{flexible\text{-}SCC}(\mathcal{D})$ is an empty set, and this happens when all nodes in the directed graph $\mathcal{M}_{\mathcal{D}}$ are not path-flexible.

Restriction of a set of node configurations. Given an LCL problem $\Pi = (\Delta, \Sigma, \mathcal{V}, \mathcal{E})$, a subset $\mathcal{S} \subseteq \mathcal{V}$ of node configurations, and a set $\mathcal{D}$ of size-2 multisets whose elements are in Σ, we define the restriction of $\mathcal{S}$ to $\mathcal{D}$ as follows.

$$\mathcal{S} \restriction_{\mathcal{D}} = \{C \in \mathcal{S} \mid \text{all size-2 sub-multisets of } C \text{ are in } \mathcal{D}\}.$$

Lemma 8 shows that if we label the two endpoints of a sufficiently long path using node configurations in $\mathcal{S} \restriction_{\mathcal{D}^*}$, where $\mathcal{D}^* \in \mathsf{flexible\text{-}SCC}(\mathcal{D}_{\mathcal{S}})$, then it is always possible to complete the labeling of the path using only node configurations in $\mathcal{S}$ in such a way that the entire labeling is correct. Specifically, consider a path $P = (v_1, v_2, \ldots, v_{d+1})$ of length $d \geq \mathsf{flexibility}(\mathcal{D}^*)$. Assume that the node configuration of v_1 is already fixed to be $C \in \mathcal{S} \restriction_{\mathcal{D}^*}$ where the half-edge $(v_1, \{v_1, v_2\})$ is labeled by $\beta \in C$ and the node configuration of v_{d+1} is already fixed to be $C' \in \mathcal{S} \restriction_{\mathcal{D}^*}$ where the half-edge $(v_{d+1}, \{v_d, v_{d+1}\})$ is labeled by $\alpha' \in C'$. Lemma 8 shows that it is possible to complete the labeling of P using only node configurations in $\mathcal{S}$, as we may label v_i using the node configuration C_i where the two half-edges $(v_i, \{v_{i-1}, v_i\})$ and $(v_i, \{v_i, v_{i+1}\})$ are labeled by α_i and β_i, for each $2 \leq i \leq d$.

▶ **Lemma 8** (Property of path-flexible strongly connected components). *Let $\mathcal{S} \subseteq \mathcal{V}$ be a set of node configurations, and let $\mathcal{D}^* \in \mathsf{flexible\text{-}SCC}(\mathcal{D}_{\mathcal{S}})$. For any choices of $C \in \mathcal{S} \restriction_{\mathcal{D}^*}$, $C' \in \mathcal{S} \restriction_{\mathcal{D}^*}$, size-2 sub-multisets $\{\alpha, \beta\} \subseteq C$, $\{\alpha', \beta'\} \subseteq C'$, and a number $d \geq \mathsf{flexibility}(\mathcal{D}^*)$, there exists a sequence $\alpha_1, C_1, \beta_1, \alpha_2, C_2, \beta_2, \ldots, \alpha_{d+1}, C_{d+1}, \beta_{d+1}$ satisfying the following conditions.*

- *First endpoint: $\alpha_1 = \alpha$, $\beta_1 = \beta$, and $C_1 = C$.*
- *Last endpoint: $\alpha_{d+1} = \alpha'$, $\beta_{d+1} = \beta'$, and $C_{d+1} = C'$.*
- *Node configurations: for $1 \leq i \leq d+1$, $\{\alpha_i, \beta_i\}$ is a size-2 sub-multiset of C_i, and $C_i \in \mathcal{S}$.*
- *Edge configurations: for $1 \leq i \leq d$, $\{\beta_i, \alpha_{i+1}\} \in \mathcal{E}$.*

Proof. By the path-flexibility of $\mathcal{D}^*$, there exists a length-d walk $(\alpha, \beta) \rightsquigarrow (\alpha', \beta')$ in $\mathcal{M}_{\mathcal{D}_\mathcal{S}}$. We fix $(\alpha_1, \beta_1) \to (\alpha_2, \beta_2) \to \cdots \to (\alpha_{d+1}, \beta_{d+1})$ to be any such walk. This implies that $\{\beta_i, \alpha_{i+1}\} \in \mathcal{E}$ for each $1 \le i \le d$. Since $\{\alpha_i, \beta_i\}$ is a size-2 multiset of $\mathcal{D}_\mathcal{S}$, there exists a choice of $C_i \in \mathcal{S}$ for each $2 \le i \le d$ such that $\{\alpha_i, \beta_i\}$ is a sub-multiset of C_i. ◀

Good sequences. Given an LCL problem $\Pi = (\Delta, \Sigma, \mathcal{V}, \mathcal{E})$ on Δ-regular trees, we say that a sequence $(\mathcal{V}_1, \mathcal{D}_1, \mathcal{V}_2, \mathcal{D}_2, \ldots, \mathcal{V}_k)$ is *good* if it satisfies the following requirements.

- $\mathcal{V}_1 = \mathsf{trim}(\mathcal{V})$. That is, we start the sequence from the result of trimming the set $\mathcal{V}$ of all node configurations in the given LCL problem $\Pi = (\Delta, \Sigma, \mathcal{V}, \mathcal{E})$.
- For each $1 \le i \le k-1$, $\mathcal{D}_i \in \mathsf{flexible\text{-}SCC}(\mathcal{D}_{\mathcal{V}_i})$. That is, $\mathcal{D}_i$ is a path-flexible strongly connected component of the automaton associated with the path-form of the LCL problem $(\Delta, \Sigma, \mathcal{V}_i, \mathcal{E})$, which is Π restricted to the set of node configurations $\mathcal{V}_i$.
- For each $2 \le i \le k$, $\mathcal{V}_i = \mathsf{trim}(\mathcal{V}_{i-1} \restriction_{\mathcal{D}_{i-1}})$. That is, $\mathcal{V}_i$ is the result of taking the restriction of the set of node configurations $\mathcal{V}_{i-1}$ to $\mathcal{D}_{i-1}$ and then performing a trimming.
- $\mathcal{V}_k \neq \emptyset$. That is, we require that the last set of node configurations is non-empty.

It is straightforward to see that $\mathcal{V}_1 \supseteq \mathcal{V}_2 \supseteq \cdots \supseteq \mathcal{V}_k$ since $\mathcal{V}_i = \mathsf{trim}(\mathcal{V}_{i-1} \restriction_{\mathcal{D}_{i-1}})$ is always a subset of $\mathcal{V}_{i-1}$. Similarly, we also have $\mathcal{D}_1 \supseteq \mathcal{D}_2 \supseteq \cdots \supseteq \mathcal{D}_{k-1}$, as $\mathcal{D}_i \in \mathsf{flexible\text{-}SCC}(\mathcal{D}_{\mathcal{V}_i})$ is a subset of $\mathcal{D}_{\mathcal{V}_i}$ and $\mathcal{D}_{\mathcal{V}_i}$ is a subset of $\mathcal{D}_{i-1}$ due to the definition $\mathcal{V}_i = \mathsf{trim}(\mathcal{V}_{i-1} \restriction_{\mathcal{D}_{i-1}})$.

Depth of an LCL problem. We define the depth d_Π of an LCL problem $\Pi = (\Delta, \Sigma, \mathcal{V}, \mathcal{E})$ on Δ-regular trees as follows. If there is no good sequence, then we set $d_\Pi = 0$. If there is a good sequence $(\mathcal{V}_1, \mathcal{D}_1, \mathcal{V}_2, \mathcal{D}_2, \ldots, \mathcal{V}_k)$ for each positive integer k, then we set $d_\Pi = \infty$. Otherwise, we set d_Π as the largest integer k such that there is a good sequence $(\mathcal{V}_1, \mathcal{D}_1, \mathcal{V}_2, \mathcal{D}_2, \ldots, \mathcal{V}_k)$. In the full version [3] of the paper, we prove the following results.

▶ **Theorem 9** (Characterization of complexity classes). *Let $\Pi = (\Delta, \Sigma, \mathcal{V}, \mathcal{E})$ be an* LCL *problem on Δ-regular trees. We have the following.*

- *If $d_\Pi = 0$, then Π is unsolvable in the sense that there exists a tree of maximum degree Δ such that there is no correct solution of Π on this tree.*
- *If $d_\Pi = k$ is a positive integer, then the optimal round complexity of Π is $\Theta(n^{1/k})$.*
- *If $d_\Pi = \infty$, then Π can be solved in $O(\log n)$ rounds.*

▶ **Theorem 10** (Complexity of the characterization). *There is a polynomial-time algorithm $\mathcal{A}$ that computes d_Π for any given* LCL *problem $\Pi = (\Delta, \Sigma, \mathcal{V}, \mathcal{E})$ on Δ-regular trees. If $d_\Pi = k$ is a positive integer, then $\mathcal{A}$ also outputs a description of an $O(n^{1/k})$-round algorithm for Π. If $d_\Pi = \infty$, then $\mathcal{A}$ also outputs a description of an $O(\log n)$-round algorithm for Π.*

In Theorem 9, all upper bounds hold in the CONGEST model, and all lower bounds hold in the LOCAL model. For example, if $d_\Pi = 5$, then Π can be solved in $O(n^{1/5})$ rounds in the CONGEST model, and there is a matching lower bound $\Omega(n^{1/5})$ in the LOCAL model. The distributed algorithms returned by the polynomial-time algorithm $\mathcal{A}$ in Theorem 10 also work in the CONGEST model. We note that there are several natural definitions of unsolvability of an LCL w.r.t. a given graph class that are different from the one in Theorem 9, see [23].

References

1 Yehuda Afek, Shay Kutten, and Moti Yung. The local detection paradigm and its application to self-stabilization. *Theor. Comput. Sci.*, 186(1-2):199–229, 1997. doi:10.1016/S0304-3975(96)00286-1.

2 Alkida Balliu, Sebastian Brandt, Yi-Jun Chang, Dennis Olivetti, Mikaël Rabie, and Jukka Suomela. The distributed complexity of locally checkable problems on paths is decidable. In *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 262–271. ACM Press, 2019. doi:10.1145/3293611.3331606.

3 Alkida Balliu, Sebastian Brandt, Yi-Jun Chang, Dennis Olivetti, Jan Studenỳ, and Jukka Suomela. Efficient classification of local problems in regular trees. *arXiv preprint*, 2022. `arXiv:2202.08544`.

4 Alkida Balliu, Sebastian Brandt, Yuval Efron, Juho Hirvonen, Yannic Maus, Dennis Olivetti, and Jukka Suomela. Classification of distributed binary labeling problems. In *Proc. 34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *LIPIcs*, pages 17:1–17:17. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPIcs.DISC.2020.17`.

5 Alkida Balliu, Sebastian Brandt, Juho Hirvonen, Dennis Olivetti, Mikaël Rabie, and Jukka Suomela. Lower bounds for maximal matchings and maximal independent sets. In *Proc. 60th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2019)*, pages 481–497. IEEE, 2019. `doi:10.1109/FOCS.2019.00037`.

6 Alkida Balliu, Sebastian Brandt, Fabian Kuhn, and Dennis Olivetti. Improved distributed lower bounds for MIS and bounded (out-)degree dominating sets in trees. In *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 283–293. ACM, 2021. `doi:10.1145/3465084.3467901`.

7 Alkida Balliu, Sebastian Brandt, Fabian Kuhn, and Dennis Olivetti. Deterministic δ-coloring plays hide-and-seek. In *Proc. 54th Annual ACM SIGACT Symposium on Theory of Computing (STOC 2022)*. ACM, 2022.

8 Alkida Balliu, Sebastian Brandt, and Dennis Olivetti. Distributed lower bounds for ruling sets. In *Proc. 61st IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 365–376, 2020. `doi:10.1109/FOCS46700.2020.00042`.

9 Alkida Balliu, Sebastian Brandt, Dennis Olivetti, Jan Studený, Jukka Suomela, and Aleksandr Tereshchenko. Locally checkable problems in rooted trees. In *Proc. 40th ACM Symposium on Principles of Distributed Computing (PODC 2021)*, pages 263–272. ACM Press, 2021. `doi:10.1145/3465084.3467934`.

10 Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and Jukka Suomela. How much does randomness help with locally checkable problems? In *Proc. 39th ACM Symposium on Principles of Distributed Computing (PODC 2020)*, pages 299–308. ACM Press, 2020. `doi:10.1145/3382734.3405715`.

11 Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and Jukka Suomela. Almost global problems in the LOCAL model. *Distributed Computing*, 34:259–281, 2021. `doi:10.1007/s00446-020-00375-2`.

12 Alkida Balliu, Keren Censor-Hillel, Yannic Maus, Dennis Olivetti, and Jukka Suomela. Locally checkable labelings with small messages. In *35th International Symposium on Distributed Computing, DISC 2021*, volume 209 of *LIPIcs*, pages 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.DISC.2021.8`.

13 Alkida Balliu, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Dennis Olivetti, and Jukka Suomela. New classes of distributed time complexity. In *Proc. 50th ACM Symposium on Theory of Computing (STOC 2018)*, pages 1307–1318. ACM Press, 2018. `doi:10.1145/3188745.3188860`.

14 Alkida Balliu, Juho Hirvonen, Dennis Olivetti, and Jukka Suomela. Hardness of minimal symmetry breaking in distributed computing. In *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 369–378. ACM Press, 2019. `doi:10.1145/3293611.3331605`.

15 Sebastian Brandt. An automatic speedup theorem for distributed problems. In *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 379–388. ACM, 2019. `doi:10.1145/3293611.3331611`.

16 Sebastian Brandt, Yi-Jun Chang, Jan Grebík, Christoph Grunau, Václav Rozhoň, and Zoltán Vidnyánszky. Local problems on trees from the perspectives of distributed algorithms, finitary factors, and descriptive combinatorics. In *13th Innovations in Theoretical Computer Science Conference, ITCS 2022*, volume 215 of *LIPIcs*, pages 29:1–29:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPIcs.ITCS.2022.29`.

17 Sebastian Brandt, Orr Fischer, Juho Hirvonen, Barbara Keller, Tuomo Lempiäinen, Joel Rybicki, Jukka Suomela, and Jara Uitto. A lower bound for the distributed Lovász local lemma. In *Proc. 48th ACM Symposium on Theory of Computing (STOC 2016)*, pages 479–488. ACM Press, 2016. doi:10.1145/2897518.2897570.

18 Sebastian Brandt, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Patric R. J. Östergård, Christopher Purcell, Joel Rybicki, Jukka Suomela, and Przemysław Uznański. LCL problems on grids. In *Proc. 36th ACM Symposium on Principles of Distributed Computing (PODC 2017)*, pages 101–110. ACM Press, 2017. doi:10.1145/3087801.3087833.

19 Sebastian Brandt and Dennis Olivetti. Truly tight-in-Δ bounds for bipartite maximal matching and variants. In *Proc. 39th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 69–78, 2020. doi:10.1145/3382734.3405745.

20 Yi-Jun Chang. The complexity landscape of distributed locally checkable problems on trees. In *Proc. 34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *LIPIcs*, pages 18:1–18:17. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.18.

21 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An exponential separation between randomized and deterministic complexity in the LOCAL model. *SIAM J. Comput.*, 48(1):122–143, 2019. doi:10.1137/17M1117537.

22 Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the LOCAL model. *SIAM J. Comput.*, 48(1):33–69, 2019. doi:10.1137/17M1157957.

23 Yi-Jun Chang, Jan Studený, and Jukka Suomela. Distributed graph problems through an automata-theoretic lens. In *Proc. 28th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2021)*, LNCS. Springer, 2021. arXiv:2002.07659.

24 Kai-Min Chung, Seth Pettie, and Hsin-Hao Su. Distributed algorithms for the lovász local lemma and graph coloring. *Distributed Comput.*, 30(4):261–280, 2017. doi:10.1007/s00446-016-0287-6.

25 Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Inf. Control.*, 70(1):32–53, 1986. doi:10.1016/S0019-9958(86)80023-7.

26 Manuela Fischer. Improved deterministic distributed matching via rounding. In *Proceedings of the 31st International Symposium on Distributed Computing (DISC 2017)*, pages 17:1–17:15, 2017. doi:10.4230/LIPIcs.DISC.2017.17.

27 Manuela Fischer and Mohsen Ghaffari. Sublogarithmic distributed algorithms for lovász local lemma, and the complexity hierarchy. In *Proc. 31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *LIPIcs*, pages 18:1–18:16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.DISC.2017.18.

28 Pierre Fraigniaud, Marc Heinrich, and Adrian Kosowski. Local conflict coloring. In *Proc. 57th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 625–634, 2016. doi:10.1109/FOCS.2016.73.

29 Mohsen Ghaffari and Fabian Kuhn. Deterministic distributed vertex coloring: Simpler, faster, and without network decomposition. In *Proc. 62nd IEEE Annual Symposium on Foundations of Computer Science (FOCS 2021)*, 2021.

30 Mohsen Ghaffari and Hsin-Hao Su. Distributed Degree Splitting, Edge Coloring, and Orientations. In *Proc. 28th ACM-SIAM Symposium on Discrete Algorithms (SODA 2017)*, pages 2505–2523. Society for Industrial and Applied Mathematics, 2017. doi:10.1137/1.9781611974782.166.

31 Christoph Grunau, Václav Rozhoň, and Sebastian Brandt. The landscape of distributed complexities on trees, 2021. arXiv:2202.04724.

32 Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992. doi:10.1137/0221015.

33 Michael Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *SIAM Journal on Computing*, 15(4):1036–1053, 1986. doi:10.1137/0215074.

34 Yannic Maus and Tigran Tonoyan. Local conflict coloring revisited: Linial for lists. In *34th International Symposium on Distributed Computing, DISC 2020*, volume 179 of *LIPIcs*, pages 16:1–16:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.16.

35 Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *Proc. 26th Annual Symposium on Foundations of Computer Science (FOCS 1985)*, pages 478–489. IEEE, 1985. doi:10.1109/SFCS.1985.43.

36 Moni Naor. A lower bound on probabilistic algorithms for distributive ring coloring. *SIAM J. Discret. Math.*, 4(3):409–412, 1991. doi:10.1137/0404036.

37 Moni Naor and Larry J. Stockmeyer. What can be computed locally? *SIAM J. Comput.*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

38 Dennis Olivetti. Round Eliminator: a tool for automatic speedup simulation, 2020. URL: https://github.com/olidennis/round-eliminator.

39 Alessandro Panconesi and Romeo Rizzi. Some simple distributed algorithms for sparse networks. *Distributed Computing*, 14(2):97–100, 2001. doi:10.1007/PL00008932.

40 Václav Rozhoň and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proc. 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC 2020)*, pages 350–363. ACM, 2020. doi:10.1145/3357713.3384298.

41 Jan Studený and Aleksandr Tereshchenko. Rooted tree classifier, 2021. URL: https://github.com/jendas1/rooted-tree-classifier.

42 J. J. Sylvester. On subvariants, i.e. semi-invariants to binary quantics of an unlimited order. *American Journal of Mathematics*, 5(1):79–136, 1882. URL: http://www.jstor.org/stable/2369536.

Exponential Speedup over Locality in MPC with Optimal Memory

Alkida Balliu
Gran Sasso Science Institute, L'Aquila, Italy

Sebastian Brandt
CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

Manuela Fischer
ETH Zürich, Switzerland

Rustam Latypov
Aalto University, Espoo, Finland

Yannic Maus
TU Graz, Austria

Dennis Olivetti
Gran Sasso Science Institute, L'Aquila, Italy

Jara Uitto
Aalto University, Espoo, Finland

Abstract

Locally Checkable Labeling (LCL) problems are graph problems in which a solution is correct if it satisfies some given constraints in the local neighborhood of each node. Example problems in this class include maximal matching, maximal independent set, and coloring problems. A successful line of research has been studying the complexities of LCL problems on paths/cycles, trees, and general graphs, providing many interesting results for the LOCAL model of distributed computing. In this work, we initiate the study of LCL problems in the low-space Massively Parallel Computation (MPC) model. In particular, on forests, we provide a method that, given the complexity of an LCL problem in the LOCAL model, automatically provides an exponentially faster algorithm for the low-space MPC setting that uses optimal global memory, that is, truly linear.

While restricting to forests may seem to weaken the result, we emphasize that all known (conditional) lower bounds for the MPC setting are obtained by lifting lower bounds obtained in the distributed setting *in tree-like networks* (either forests or high girth graphs), and hence the problems that we study are challenging already on forests. Moreover, the most important technical feature of our algorithms is that they use optimal global memory, that is, memory linear in the number of edges of the graph. In contrast, most of the state-of-the-art algorithms use more than linear global memory. Further, they typically start with a dense graph, sparsify it, and then solve the problem on the residual graph, exploiting the relative increase in global memory. On forests, this is not possible, because the given graph is already as sparse as it can be, and using optimal memory requires new solutions.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed computing, Locally checkable labeling problems, Trees, Massively Parallel Computation, Sublinear memory

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.9

Related Version *Full Version*: https://arxiv.org/abs/2208.09453

Funding *Rustam Latypov*: Supported in part by the Academy of Finland, Grant 334238.

1 Introduction

The Massively Parallel Computation (MPC) model, introduced in [42] and later refined by [2, 12, 39], is a mathematical abstraction of modern data processing platforms such as MapReduce [28], Hadoop [55], Spark [56], and Dryad [41]. Recently, tremendous progress has been made on fundamental *graph problems* in this model, such as maximal independent set (MIS), maximal matching (MM) [37, 25], and coloring problems [19, 27]. All these

© Alkida Balliu, Sebastian Brandt, Manuela Fischer, Rustam Latypov, Yannic Maus, Dennis Olivetti, and Jara Uitto;
licensed under Creative Commons License CC-BY 4.0

problems, and many others, fall under the umbrella of *Locally Checkable* problems, in which the feasibility of a solution can be checked by inspecting local neighborhoods. They also serve as abstractions for fundamental primitives in large-scale graph processing and have recently gained a lot of attention [9, 7, 27, 29, 3, 18, 32]. *Locally checkable labelings (LCLs)* are locally checkable problems restricted to constant degree graphs. A more formal definition of LCLs is deferred to Section 2.

LCLs have been a rich source of research in various models of computation, because they can be seen as a starting point to understand locally checkable problems in general, and this holds independently of the model. For example, in the distributed setting, techniques developed to understand LCLs [16] have then been used to prove lower bounds in the unbounded degree setting, which the LCL setting does not include, e.g., for the the maximal independent set problem, or the Δ-coloring problem [4, 6, 5]. In the distributed LOCAL model of computing, a lot is known about LCLs: for example, if the graph on which we want to solve the problem is a tree, then there is a discrete set of possible complexities, and in some cases, given an LCL, we can even automatically decide its distributed time complexity. Our goal is to bring to the parallel setting, and in particular to the MPC model, the knowledge that researchers developed about LCLs in the distributed setting, while also developing new techniques that can be used in the parallel setting. We show that, on forests, the mere knowledge of what is the distributed complexity of a problem is enough to obtain blazingly fast algorithms in the MPC setting. In particular, we obtain MPC algorithms that are *exponentially faster* than the best distributed ones. We summarize our main result.

The complexity of any LCL problem on forests in the MPC model is exponentially lower than its distributed complexity, even when using optimal memory bounds.

More in detail, in our work, we solve LCL problems in forests in the most restrictive *low-space* MPC model with *linear* total memory, which is the most *scalable* variant of the MPC model. Our results provide an automatic method that, for all LCL problems, yields an algorithm that solves the given problem *exponentially* faster than its optimal distributed counterpart. The resulting algorithms are *component-stable* [35, 26], which implies that the solutions in individual connected components are independent of the other components. Our results are in some sense optimal: for problems that in the LOCAL model can be solved in $n^{o(1)}$, finding more-than-exponentially faster component-stable algorithms would violate the widely-believed 1 vs. 2 cycle conjecture in the MPC setting.

Why do we care about trees and forests? All known conditional lower bounds[1] for problems in the MPC setting are derived by lifting lower bounds that hold in the LOCAL model of distributed computing [35, 26]. Most of the lower bounds known in the LOCAL model are actually proved either on trees or on high-girth graphs (where the neighborhood of each node corresponds to a tree): see, e.g., [44, 5, 4, 6, 16]. It follows that essentially all the conditional lower bounds known in the MPC setting already hold on forests[2]. Despite this fact, with a few exceptions, there is no work on upper bounds on forests in the MPC model – a gap we aim to fill.

[1] Proving unconditional lower bounds for the MPC model would imply a major breakthrough in circuit complexity and seems out of reach [53].

[2] As lifting lower bounds from the LOCAL model to the MPC model requires hereditary graph classes one cannot immediately lift a lower bound in the LOCAL model that holds on trees. Instead, a lower bound in the LOCAL model on trees implies the same lower bound in the LOCAL model for forests which can then be lifted to a lower bound for MPC algorithms on forests.

Moreover, understanding the complexity of problems on trees has been already shown to be essential in the LOCAL model: it is typically the case that interesting problems are already challenging on trees, and often even in regular balanced trees of small degree. In fact, most lower bounds known in the LOCAL model hold exactly in this setting. Due to the lifting, the same statement adapted to forests is true for all recent MPC lower bounds. Hence, to decrease the relevance of trees and forests, we either need completely new lower bound techniques in the LOCAL model coupled with completely new lifting theorems, or completely new lower bound techniques for the MPC model.

At first glance it may seem that our results are easy to achieve, because we restrict to forests. Conversely, we would like to emphasize that many state-of-the-art algorithms for problems like MIS and coloring work as follows [37, 25]: start with a dense graph which requires a lot of memory to store, sparsify it, and then use the freed global memory to solve the problem faster on the sparsified part. On forests, this is not possible, because the given graph is already as sparse as it can be.

The MPC Model. In the MPC model, we have M machines who communicate in an all-to-all fashion. We focus on problems where the input is modeled as a graph with n vertices, m edges and maximum degree Δ; we call this graph the *input graph*. Each node has a unique ID of size $b = O(\log n)$ bits from a domain $\{1, 2, \dots, N\}$, where $N = \text{poly}(n)$. Each node and its incident edges are hosted on a machine(s) with $S = O(n^\delta)$ *local memory*, where $\delta \in (0, 1)$ and the units of memory are *words* of $O(\log n)$ bits. When the local memory is bounded by $O(n^\delta)$, the model is called *low-space* (or *sublinear*). The number of machines is chosen such that $M = m/S = \Theta(m/n^\delta)$. For trees, where $m = \Theta(n)$, this results in $\Theta(n^{1-\delta})$ machines, that is, a *total memory* (or *global memory*) of $M \cdot S = \Theta(n)$. For simplicity[3], we assume that each machine i simulates one virtual machine for each node and its incident edges that i hosts, such that the local memory restriction becomes that no virtual machine can use more than $O(n^\delta)$ memory.

During the execution of an MPC algorithm, computation is performed in synchronous, fault-tolerant rounds. In each round, every machine performs some (unbounded) computation on the locally stored data, then sends/receives messages to/from any other machine in the network. Each message is sent to exactly one other machine specified by the sending machine. All messages sent and received by each machine in each round, as well as the output, have to fit into local memory. The time complexity is the number of rounds it takes to solve a problem. Upon termination, each node (resp. its hosting machine) must know its own part of the solution. For example in the case of node-coloring, the machine hosting node u must decide on the color of u upon termination of the algorithm.

Unlike in most other works, our algorithms employ $O(m + n)$ words of total memory, which is the strictest possible as it is only enough to store a constant number of copies of the input graph. Note that if we were to allow superlinear $O(m^{1+\delta})$ global memory in our constant-degree setting, many LOCAL algorithms with complexity $O(\log n)$ could be trivially sped up exponentially in the low-space MPC model by applying the well-known graph exponentiation technique by Lenzen and Wattenhofer [45]. A crucial challenge that comes with the linear global memory restriction is that only a small fraction of $n^{1-\delta}$ of the (virtual) machines can simultaneously utilize all of their available local memory. Thus, with

[3] In practice, it is assumed that the virtual machines can be shuffled between physical machines, such that the sum of the memory of the virtual machines hosted on any single physical machine is $O(n^\delta)$.

strictly linear global memory we are forced to develop new techniques which must avoid gathering local neighborhoods, i.e., fundamentally divert from direct simulations of message passing algorithms.

1.1 The Distributed Complexity Landscapes

In the last decade, there has been tremendous progress in understanding the complexities of LCLs in various models of distributed and parallel computing. A prime example is the LOCAL model [46], where the input graph corresponds to a message passing system, and the nodes must output their part of the solution according only to local information about the graph. Another example is the CONGEST model, which is a LOCAL model variant where the message size is restricted to $O(\log n)$ bits [52]. A curious fact about LCLs in the distributed setting is the existence of *complexity gaps*, that is, some complexities are not possible at all. For example, it is known that there are no LCLs with a distributed time complexity in the LOCAL and CONGEST model that lies between $\omega(\log^* n)$ and $o(\log n)$. In these two models, the *whole* complexity landscape of LCL problems is now understood for some important graph families. For instance, a rich line of work [50, 20, 22, 11, 8, 3, 18] recently came to an end when a complexity gap between $\omega(1)$ and $o(\log^* n)$ was proved [40], completing the randomized/deterministic complexity landscape of LCL problems in the LOCAL model for trees. In the CONGEST model, the authors of [9] showed that, on trees, the complexity of an LCL problem is asymptotically equal to its complexity in the LOCAL model, whereas the same does not hold in general graphs. In the randomized/deterministic LOCAL and CONGEST models, recent work showed that the complexity landscapes of LCL problems for rooted regular trees are fully understood [7], while the complexity landscapes of LCL problems in the LOCAL model for rings and tori have already been known for some while [17]. Even for general (constant-degree) graphs, the LOCAL complexity landscape of LCL problems is almost fully understood [50, 16, 21, 22, 36, 30, 34, 10, 8, 54, 33], only missing a small part of the picture related to the randomized complexity of Lovász Local Lemma (LLL).

In the case of trees, for deterministic algorithms in the LOCAL model, it is known that there is a discrete set of possible complexities, that we divide into four categories:

- **Tiny regime**: contains the complexities $O(1)$ and $\Theta(\log^* n)$.
  - Example problems: maximal independent set, maximal matching, $(\Delta + 1)$-vertex coloring[4], $(2\Delta - 1)$-edge coloring, and trivial problems (e.g., all nodes must output 0).
- **Mid regime**: contains the complexity $\Theta(\log n)$.
  - Example problems: sinkless orientation [16], 3-coloring, and Δ-coloring.
- **High regime**: contains the complexities $\Theta(n^{1/k})$, for all $k \in \mathbb{N}$.
  - Example problems: 2-coloring and $2\frac{1}{2}$-coloring [22].

Moreover, it is known that randomness can help only in the mid regime, and in particular that *some* problems requiring $\Theta(\log n)$ for deterministic algorithms have randomized complexity $\Theta(\log \log n)$, which constitutes our fourth category – **Low regime**. Problems residing in the low regime include sinkless orientation and Δ-coloring.

On forests, the complexity landscape in the LOCAL model is the same as on trees. While this is intuitively evident, it can also be shown formally using an analogous approach to the one used in the proof of [40, Lemma 3.3].

[4] We denote the maximum degree of the graph by Δ.

1.2 Our Contributions

Our main contribution is showing that, given any LCL problem (see Definition 4) on trees that has deterministic (resp. randomized) complexity T in the LOCAL model, we can automatically obtain an MPC algorithm with deterministic (resp. randomized) complexity $O(\log T)$ on forests. In particular, we prove the following.

▶ **Theorem 1.** *Consider an LCL problem on trees with deterministic time complexity $f(n)$ and randomized time complexity $g(n)$ in the LOCAL model. This problem has deterministic time complexity $O(\log f(n))$ and randomized time complexity $O(\log g(n))$ in the low-space MPC model on forests using optimal $O(m+n)$ words of global memory. The provided algorithms are component-stable.*

Put differently, a problem in the LOCAL model can only have a deterministic complexity $f(n) \in \{\Theta(1), \Theta(\log^* n), \Theta(\log n)\} \cup \{\Theta(n^{1/k}) \mid k \in \mathbb{N}\}$, and we show that it is enough to know the asymptotic value of $f(n)$ in order to obtain a deterministic MPC algorithm with complexity $O(\log(f(n))) \in \{O(1), O(\log \log^* n), O(\log \log n), O(\log n)\}$.

Moreover, it is known that for all $f(n) \notin \Theta(\log n)$, the LOCAL randomized complexity of the problem is the same as the deterministic one. Instead, for $f(n) \in \Theta(\log n)$, the LOCAL randomized complexity $g(n)$ can be either $\Theta(\log n)$ or $\Theta(\log \log n)$. If it is $\Theta(\log \log n)$, then we provide an MPC algorithm with randomized complexity $O(\log \log \log n)$. If we dismiss the component-stability requirement, we can obtain the same $O(\log \log \log n)$ runtime with a deterministic MPC algorithm.

▶ **Theorem 2.** *Consider an LCL problem on trees with randomized time complexity $g(n) = \Theta(\log \log n)$ in the LOCAL model. This problem has deterministic time complexity $O(\log \log \log n)$ in the low-space MPC model on forests using optimal $O(m+n)$ words of global memory. This algorithm is component-unstable.*

By [35, 26], we know that Theorem 1 is in some sense optimal: if a problem requires T deterministic rounds in the LOCAL model, then it requires $\Omega(\min\{\log T, \log \log n\})$ rounds in the low-space MPC setting for component-stable algorithms, assuming that the infamous 1 vs. 2 cycle conjecture holds [12, 35, 53]. In contrast, Theorem 2 shows that one can break the conditional lower bound of $\Omega(\log \log n)$ for deterministic MPC algorithms for all LCL problems in the aforementioned class by diverting to component-unstable algorithms. Achieving the same result even for a single problem without dismissing the component-stability requirement would be a major breakthrough, as it would falsify the conjecture.

As a subroutine for solving all problems that belong to the high regime in $O(\log n)$ MPC rounds, we also develop an $O(\log n)$ round MPC algorithm for rooting a forest. This rooting algorithm is component-stable, and may be of independent interest, since it is also compatible with arbitrary degrees.

Additional observations. There is a long line of research that provided algorithms for MPC that are exponentially faster than the best algorithms for the LOCAL model. Most existing results achieved these speedup results by using additional global memory, that is, $\omega(m)$ words [13, 32, 27, 26]. We emphasize that, deviating from the usual approach, all of our results use *optimal* MPC parameters, in the sense that we work in the low-space setting with $O(n^\delta)$ words of local memory and $O(m+n)$ words of global memory.

Hence, our contribution is twofold, on the one hand we prove that we can indeed achieve this exponential speedup for all LCLs, while on the other hand we show that this exponential speedup can be achieved without requiring any additional memory. Furthermore, graph

problems in trees and forests are widely unexplored, despite their central role that we have already elaborated on. It is known that a 4-coloring, MIS, and maximal matching can be found in $O(\log \log n)$ rounds [32]. However, the coloring result heavily relies on randomness and the MIS and matching results require a (small) overhead in the total memory. To compare, our results deterministically yield a 3-coloring in $O(\log \log n)$ rounds with linear total memory. It is not clear whether randomness can even help in the case of 3-coloring, which is a significant difference to the case of 4-coloring. Furthermore, it is not clear whether the previous approaches to MIS and matching can be extended to work deterministically with the same runtime and with linear total memory. While the previous work is designed for arbitrary degree graphs, it is not clear whether the algorithms could be tuned to work faster with constant degrees.

Open Questions. In the tiny regime, our results extend to general graphs (see Theorem 5). In the low regime, our results extend to general graphs if we allow slightly more global memory (see Theorem 6.5 in the full version). Once we reach the mid regime, i.e., logarithmic distributed complexities, we do not know the behaviour in general graphs. This leads to an interesting open question. As mentioned, the asymptotic complexity of any problem on trees is identical in the LOCAL and CONGEST model, and the same is true (modulo the exact complexity of the LLL in both models) on general graphs as long as the complexity is sublogarithmic [9]. However, there is a an exponential separation between the models for complexities that are at least logarithmic [9]. Does such a separation between the complexity of an LCL in the LOCAL model and the MPC model also hold for large complexities? Here, of course, we would want to have a doubly exponential separation.

Interestingly, current conditional lower bounds for the MPC model cannot prove MPC lower bounds that are $\omega(\log \log n)$. So, while our results in the high regime show that any problem on forests can be solved in $O(\log n)$ rounds in the MPC model, it remains unclear whether we cannot improve on this bound, even without falsifying the 1 vs. 2 cycle conjecture.

Component-stability. The term of a *component-stable* MPC algorithm has been introduced in [35] in the context of lifting distributed lower bounds to the MPC setting. By their definition, informally, an algorithm is component-stable if the output of a node does not change if other connected components in the graph are altered (see Definition 11).

While initially believed that it might be an artifact of their lifting techniques, Czumaj, Davies and Parter [26] showed the contrary, i.e., they showed that *component-unstable* algorithms can beat the conditional lower bounds of [35]. Their results hold assuming their revised definition of component-stability, which is argued to be more robust (see Definition 13). Under their definition, it is not strictly easier nor harder to design algorithms to be component-stable, as compared to the definition of [35]. The main difference is that they allow the output of component-stable algorithms to depend on the total number of nodes in the graph and the maximum degree. In our work, we adopt the revised definition of component-stability [26]. See Appendix A and the discussion therein for further details.

1.3 Challenges & Key Techniques

We now provide an overview of the challenges that we had to tackle in order to prove our results, and a very high level explanation of the key techniques that we used to solve them.

The tiny regime serves as a good warm-up to see why using an optimal amount of global memory is difficult. The most technically involved part is the high regime, where we obtain an $O(\log n)$-time MPC algorithm for any LCL problem.

Graph Exponentiation. A reoccurring challenge for all regimes lies in respecting the linear global memory, which roughly means that on average, every node can use only a constant amount of memory. This is particularly unfortunate because almost all recent MPC results – and in particular all that achieve exponential speedups – rely on the memory-intense *graph exponentiation* technique [45]. Informally, this technique enables a node to gather its 2^k-hop neighborhood in k communication rounds. Doing this in parallel for every node in the graph results in a Δ^{2^k} overhead in global memory. For this technique to be useful, k has to be $\omega(1)$, yielding a non-constant multiplicative increase in the global memory requirement. In order to use this technique but not violate linear global memory, we develop new solutions that are discussed in the following paragraphs.

Tiny regime $f(n) = \Theta(1)$ and $f(n) = \Theta(\log^* n)$. Handling the $\Theta(1)$ complexity is trivial, since any LOCAL algorithm for LCLs can be simulated in the MPC setting. For the $\Theta(\log^* n)$ class, it is known from prior work that all problems can be solved in the LOCAL model in a very specific way: reduce to the problem of computing a distance-k coloring with a small enough number of colors, where k is a constant that depends on the problem. In a distance-k c-coloring, each node is assigned a color in $\{1, \ldots, c\}$ such that nodes at distance at most k have different colors. Such a coloring can be computed in $O(\log^* n)$ rounds in the LOCAL model, and it could be computed easily in the MPC setting in $O(\log \log^* n)$ rounds, by exploiting the graph exponentiation technique, if we allow an additional $O(\log^* n)$ factor overhead in the amount of global memory.

We show that this overhead is not required, by developing a novel MPC algorithm for coloring. The algorithm that we provide reduces the problem of coloring a general graph to coloring directed pseudoforests, that is, graphs where all edges are oriented and every node has at most one outgoing edge. Then, we show that in directed pseudoforests, it is possible to solve the coloring problem through a variant of graph exponentiation that only requires keeping track of a constant number of IDs. This way, the memory use of each node is constant, and the global memory is linear.

High regime $f(n) = \Theta(n^{1/k})$, for all $k \in \mathbb{N}$. We explicitly provide, for any solvable LCL, a novel algorithm that has a runtime of $O(\log n)$. Essentially, we solve each tree in the forest separately, hence we will consider trees in the following argumentation. On a high level, our algorithm first roots the tree using our $O(\log n)$-time tree rooting algorithm, and then proceeds in two phases. In the first phase, roughly speaking, the goal is to compute, for a substantial number of nodes v, the set of possible output labels that can be output at v such that the label choice can be extended to a (locally) correct solution in the subtree hanging from v. This is done in an iterative manner, proceeding from the leaves towards the root. The second phase consists of using the computed information to solve the given LCL from the root downwards.

While this outline sounds simple, there are a number of intricate challenges that require the development of novel techniques, both in the design of the algorithm and its analysis. For instance, the depth of the input tree may be $\omega(\log n)$ (which prevents us from performing the above ideas in a sequential manner), and the storage of the required completability information grows exponentially when using graph exponentiation, exceeding the available global memory. Our key technical contributions are the following.

- The design of a process that allows for interleaving graph exponentiation steps and compressing the graph (and compatibility information) such that the process is also reversible (second phase of the algorithm). The main challenge here is that multiple graph exponentiation processes executed on individual parts of the tree have to be merged, simultaneously or at different times, into one process during the execution.

- The design of a fine-tuned potential function for the analysis of the complex algorithm resulting from addressing the aforementioned issues and the highly non-sequential behavior arising from interleaving graph exponentiation steps.

Mid regime $f(n) = \Theta(\log n)$. We would wish to use the algorithm of Chang and Pettie [22] as a black box. On a very high level idea, their LOCAL algorithm uses $O(\log n)$ rounds to compute a rake-and-compress decomposition of size $O(\log n)$, which is essentially the classic H-partition by Miller and Reif [49]. Then, compatibility information of the given LCL problem is propagated layer by layer to the top, and then labels are fixed at the top and propagated down.

Applying known MPC techniques like graph exponentiation to speed up this process does not work out of the box for several reasons. First, the compatibility information they propagate grows exponentially, which creates congestion in the MPC model. Secondly, since the input graph is as sparse as it could possibly be, the direct application of graph exponentiation would violate the optimal global memory bounds we are striving for. We resolve the first issue by first observing that the compatibility information can be reduced to constant size in every iteration. The second issue is remedied by interleaving exponentiation steps with memory freeing steps in a balanced way.

Low regime $g(n) = \Theta(\log \log n)$. With an additional $O(\log n)$ factor of global memory, this result is easy to obtain. Previous work [9] has a constant time reduction to instances of size $N = \log n$, resulting in a LOCAL algorithm with runtime $\text{poly}(\log N) = \text{poly}(\log \log n)$. A straightforward application of graph exponentiation would yield an MPC algorithm with runtime $O(\log \log \log n)$. Exploiting additional global memory in this manner has been used in a similar setting in [26]. However, without the additional memory it is harder to solve the small instances in triple logarithmic time. The work around for this memory issue is to use our mid regime algorithm on the small instances, yielding a memory efficient algorithm with runtime $O(\log \log \log n)$. To the best of our knowledge there is no other paper that can efficiently deal with such occurring small instances – small instances occur also in many other problems like MIS and graph coloring – with optimal global memory.

1.4 Further Related Work

For many of the classic graph problems, simple $O(\log n)$-time MPC algorithms follow from classic literature in the LOCAL model and PRAM [1, 46, 48]. In particular in the case of bounded degree graphs, it is often straightforward to simulate algorithms from other models. However, it is usually desirable to get algorithms that run *much faster* than their LOCAL counterparts. If the MPC algorithms are given *linear* $\Theta(n)$ or even *superlinear* $\Theta(n^{1+\delta})$ local memory, fast algorithms are known for many classic graph problems.

In the sublinear (or low-space) model, [19] provided a randomized algorithm for the $(\Delta+1)$-coloring problem that, combined with the new network decomposition results [54, 33], yields an $O(\log \log \log n)$ MPC algorithm, that is exponentially faster than its LOCAL counterpart. A recent result by Czumaj, Davies, and Parter [27] provides a deterministic $O(\log \log \log n)$-time algorithm for the same problem using derandomization techniques. For many other problems, the current state of the art in the sublinear model is still far from the aforementioned exponential improvements over the LOCAL counterparts, at least in the case of general graphs. For example, the best known MIS, maximal matching, $(1 + \epsilon)$-approximation of maximum matching, and 2-approximation of minimum vertex cover algorithms run in $\widetilde{O}(\sqrt{\log \Delta} + \sqrt{\log \log n})$ time [37], whereas the best known LOCAL algorithm has a

logarithmic dependency on Δ [31]. For restricted graph classes, such as trees and graphs with small arboricity[5] α, better algorithms are known [15, 13]. Through a recent work by Ghaffari, Grunau and Jin, the current state of the art for MIS and maximal matching are $O(\sqrt{\log \alpha} \cdot \log\log \alpha + \log\log n)$-time algorithms using $\widetilde{O}(n+m)$ words of global memory [32].

As for lower bounds, [35] gave conditional lower bounds of $\Omega(\log\log n)$ for component-stable sublinear MPC algorithms for constant approximation of maximum matching and minimum vertex cover, and MIS. In addition, the authors provided a lower bound of $\Omega(\log\log\log n)$ for LLL. Their hardness results are conditioned on a widely believed conjecture in MPC about the complexity of the connectivity problem, which asks to detect the connected components of a graph. It is argued that disproving this conjecture would imply rather strong and surprising implications in circuit complexity [53]. When assuming component-stability, they also argue that all known algorithms in the literature are component-stable or can easily be made component-stable with no asymptotic increase in the round complexity. However, recent work [26] gave a separation between stable and unstable algorithms, and that some particular problems (e.g., computing an independent set of size $\Omega(n/\Delta)$) can be solved faster with unstable algorithms than with stable ones.

It is also worth discussing the complexity of rooting a tree, as it is an important subroutine in our high regime. On the randomized side, [15] gave an $O(\log d \cdot \log\log n)$ time algorithm, where d is the diameter of the graph. On the deterministic side, Coy and Czumaj [24] gave an $O(\log n)$ time algorithm using (component-unstable) derandomization methods, which is the current state of the art. In the full version we provide a totally different rooting algorithm that is also deterministic and takes $O(\log n)$ time, but is component-stable. We note that [43] uses similar techniques in a more general setting, but in $\omega(\log n)$ time.

1.5 Outline

After the formal introduction of LCL problems and other notations in Section 2, we start proving the exponential speedup for the different regimes in Theorem 1 in separate sections. In Section 3, we warm up with the tiny regime. In Section 4, we present the high level ideas for our most involved result, the exponential speedup for the high regime. We provide full proofs and handle the remaining regimes in the full version of the paper.

Some of our speedup results use a description of a distributed algorithm with the claimed runtime to obtain the speedup. In the full version of the paper, we show that such a description can be inferred merely by knowing the distributed complexity class in which the problem resides.

2 Definitions and Notation

We work with undirected, finite, simple graphs $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges such that $E \subseteq [V]^2$ and $V \cap E = \emptyset$. Let $\deg_G(v)$ denote the degree of a node v in G and let Δ denote the maximum degree of G. The distance $d_G(v, u)$ between two vertices v, u in G is the length of a shortest $v - u$ path in G; if no such path exists, we set $d_G(v, u) := \infty$. The greatest distance between any two vertices in G is the diameter of G, denoted by $\operatorname{diam}(G)$. For a subset $S \subseteq V$, we use $G[S]$ to denote the subgraph of G induced by nodes in S. Let G^k, where $k \in \mathbb{N}$, denote the k:th power of a graph G, which is another graph on the same

[5] The arboricity of a graph is the minimum number of disjoint forests into which the edges of the graph can be partitioned.

vertex set, but in which two vertices are adjacent if their distance in G is at most k. In the context of MPC, G^k is the resulting virtual graph after performing $\log k$ steps of graph exponentiation [45].

For each node v and for every radius $k \in \mathbb{N}$, we denote the k-hop (or k-radius) neighborhood of v as $N^k(v) = \{u \in V : d(v, u) \leq k\}$. The topology of a neighborhood $N^k(v)$ of v is simply $G[N^k(v)]$. However, with slight abuse of notation, we sometimes refer to $N^k(v)$ both as the node set and the subgraph induced by node set $N^k(v)$. Neighborhood topology knowledge is often referred to as vision, e.g., node v sees $N^k(v)$. In trees and forests, the number n of nodes and the number m of edges are asymptotically equal, and we may use them interchangeably throughout the paper when reasoning about global memory.

2.1 LCL Definitions

In their seminal work [50], Naor and Stockmeyer introduced the notion of a locally checkable labeling problem (LCL problem or just LCL for short). The definition they provide restricts attention to problems where nodes are labeled (such as vertex coloring problems), but they remark that a similar definition can be given for problems where edges are labeled (such as edge coloring problems). A modern way to define LCL problems that captures both of the above types of problems (and combinations thereof) labels *half-edges* instead, i.e., pairs (v, e) where e is an edge incident to vertex v. Moreover, on trees it is known that all LCL problems can be rephrased in a special form, called *node-edge-checkable LCL problems* [9, 40]. Let us first define a half-edge labeling formally, and then provide this modern LCL problem definition.

▶ **Definition 3** (Half-edge labeling). *A* half-edge *in a graph $G = (V, E)$ is a pair (v, e), where $v \in V$ is a vertex, and $e \in E$ is an edge incident to v. A half-edge (v, e) is incident to some vertex w if $v = w$. We denote the set of half-edges of G by $H = H(G)$. A* half-edge labeling *of G with labels from a set Σ is a function $g\colon H(G) \to \Sigma$.*

We distinguish between two kinds of half-edge labelings: *input labelings* that are part of the input and *output labelings* that are provided by an algorithm executed on input-labeled instances. Throughout the paper, we will assume that any considered input graph G comes with an input labeling $g_{\text{in}}\colon H(G) \to \Sigma_{\text{in}}$ and will refer to Σ_{in} as the *set of input labels*; if the considered LCL problem does not have input labels, we can simply assume that $\Sigma_{\text{in}} = \{\perp\}$ and that each node is labeled with $\perp$.

While the formal definition of a node-edge-checkable LCL (see below) appears complicated, the intuition behind it is simple: essentially, we have a list of allowed output label combinations around nodes, a list of allowed output label combinations on edges, and a list of allowed input-output label combinations, all of which a correct solution for the LCL has to satisfy.

▶ **Definition 4** (Node-edge-checkable LCL). *Let Δ be some non-negative integer constant. A* node-edge-checkable LCL *is a quintuple $\Pi = (\Sigma_{\text{in}}, \Sigma_{\text{out}}, \mathcal{N}, \mathcal{E}, g)$ where Σ_{in} and Σ_{out} are finite sets, $\mathcal{N} = \{\mathcal{N}_1, \ldots, \mathcal{N}_\Delta\}$ consists of sets $\mathcal{N}_i$ of cardinality-i multisets with elements from Σ_{out}, $\mathcal{E}$ is a set of cardinality-2 multisets with elements from Σ_{out}, and $g\colon \Sigma_{\text{in}} \to 2^{\Sigma_{\text{out}}}$ is a function mapping input labels to sets of output labels. We call $\mathcal{N}_1 \cup \cdots \cup \mathcal{N}_\Delta$ and $\mathcal{E}$ the* node constraint *and* edge constraint *of Π, respectively. Furthermore, we call each element of $\mathcal{N}$ a* node configuration*, and each element of $\mathcal{E}$ an* edge configuration*. For a node v, denote the half-edges of the form (v, e) for some edge e by $h_1^v, \ldots, h_{\deg(v)}^v$ (in arbitrary order). For an edge e, denote the half-edges of the form (v, e) for some node v by h_1^e, h_2^e (in arbitrary order).*

A correct solution for Π *on a graph* G *is a half-edge labeling* $g_{\text{out}}\colon H(G) \to \Sigma_{\text{out}}$ *such that*

1. *for each node* v*, the multiset of outputs assigned by* g_{out} *to* $h_1^v, \ldots, h_{\deg(v)}^v$ *is an element of* $\mathcal{N}_{\deg(v)}$*,*
2. *for each edge* e*, the cardinality-*2 *multiset of outputs assigned by* g_{out} *to* h_1^e, h_2^e *is an element of* $\mathcal{E}$*, and*
3. *for each half-edge* $h \in H(G)$*, we have* $g_{\text{out}}(h) \in g(\iota)$*, where* $\iota = g_{\text{in}}(h)$ *is the input label assigned to* h*.*

We say that an algorithm $\mathcal{A}$ *solves* an LCL problem Π on a graph class $\mathcal{G}$ if it provides a correct solution for Π for every $G \in \mathcal{G}$. Note that the LCL definitions above implicitly require that graph class $\mathcal{G}$ has constant degree.

3 The Tiny Regime

In this section, we show that any LCL problem on general graphs that can be solved in the LOCAL model in $O(\log^* n)$ rounds, can be solved in the MPC model in $O(\log \log^* n)$ rounds. By combining this result with known gaps in the landscape of possible complexities in the LOCAL model [21], we obtain the following result.

▶ **Theorem 5.** *Let* Π *be an LCL problem on general graphs. Assume that there is a deterministic algorithm for the LOCAL model that solves* Π *in* $o(\log n)$ *rounds, or a randomized algorithm that solves it in* $o(\log \log n)$ *rounds. Then, the problem* Π *can be solved deterministically in* $O(\log \log^* N)$ *rounds in the low-space MPC model using* $O(m+n)$ *words of global memory, where* $N = \text{poly}(n)$ *is the size of the ID space. The algorithm works even if the graph consists of disconnected components, and it is components-stable.*

The rest of this section is devoted to proving Theorem 5.

A Universal Algorithm. In the LOCAL model, it is known that, if an LCL can be solved with an algorithm A in $o(\log n)$ deterministic rounds, or in $o(\log \log n)$ randomized rounds, then it can also be solved with a deterministic algorithm A' that requires just $O(\log^* n)$ rounds [21]. In order to prove this result, [21] shows how to convert any such algorithm A into an algorithm A' that works as follows (for some constant k that depends on the problem Π and the algorithm A):

1. Compute a distance-k $O(\Delta^{2k})$-coloring of the graph;
2. Run a k-round algorithm B that uses the computed coloring to produce the final output.

In [21] is shown that the constant k, and the k-round algorithm B, can be mechanically determined from the original algorithm A. The runtime of algorithm A' is $O(\log^* n)$ rounds since this is the runtime for the first step, while the second step only requires constant time.

Why it Works. The high-level purpose of computing the coloring in Item 1 is to provide new identifiers at the nodes that are unique up to distance k and come from a much smaller space than the original identifiers (that are part of the setting in the LOCAL model). Roughly speaking, this ensures that the k-hop view of any node that interprets the computed colors as identifiers is consistent with the node living in a constant-sized graph (with a constant-sized identifier space).

In [21], it is argued why this approach works, and on a high level, the reason can be summarized as follows. For some sufficiently large constant k, algorithm A can be executed on all graphs of a suitable constant size with a runtime of just k rounds. Since each node of the original graph executing this k-round algorithm cannot distinguish between living in

the original graph with the generated new identifiers and living in (a suitable) one of these constant-sized graphs (on all of which the algorithm is correct), the k-round algorithm must also be correct on the (much larger) original graph. This is just a high-level sketch of the proof presented in [21]; there are a number of intricate details that have to be taken care of and are explained in [21].

How We Proceed. For our purpose, we do not actually need to know the details of [21] on how A' is constructed as a function of A, and we just use the following statement that comes from [21]: if the problem Π can be solved in $o(\log n)$ deterministic rounds or $o(\log\log n)$ randomized rounds, then it can also be solved in $O(\log^* n)$ deterministic rounds using an algorithm that first applies Item 1 and then applies Item 2. In fact, in our case, we are not even given the algorithm A as input: we just know that the problem can be solved in $o(\log n)$ deterministic or $o(\log\log n)$ randomized rounds, but we are not given an algorithm A with such a complexity. Hence, we cannot apply the construction of [21] directly.

In Section 7 of the full version, we show that this is not an issue, in the sense that, if an algorithm exists, then it can be found by brute force. To show that, we use the following two important ingredients presented in [50]:

- Any constant time algorithm that solves an LCL in the LOCAL model can be transformed into an algorithm that does not require nodes to have IDs.
- For every k, it is decidable whether there exists a k-round algorithm that solves a given problem in a setting where we do not have IDs and we are given a (suitable) distance-k coloring. The reason is that, in this setting, there are only a finite number of possible algorithm candidates (and they can be enumerated), and given a candidate, it is possible to check if it constitutes a correct algorithm by using a centralized offline procedure.

We use the above ingredients as follows. If we just know that Π can be solved in $o(\log n)$ deterministic rounds or $o(\log\log n)$ randomized rounds, even if no algorithm is given, we can use [21] to claim that there exists a k for which there is a k-round algorithm B that solves Π given a distance-k coloring, and then use the first ingredient to claim that this algorithm does not need the presence of IDs. Finally, we use the second ingredient to say that if we try increasing values of k, we are going to find the algorithm B that we need.

From the above discussion, in order to prove Theorem 5, we only need to show how to compute a distance-k $O(\Delta^{2k})$-coloring in $O(\log\log^* n)$ deterministic MPC rounds.

3.1 LOCAL Algorithm

We start by presenting an algorithm for computing such a coloring in the LOCAL model. While computing such a coloring in the LOCAL model is easy, we present an algorithm amenable to be converted into a faster MPC algorithm. This algorithm is not new: it has been already presented in [38, 51], and we report it here, with minor modifications, for completeness.

▶ **Lemma 6.** *For any constant k, the distance-k $O(\Delta^{2k})$-coloring problem on general graphs can be solved in the LOCAL model with a deterministic algorithm running in $O(\log^* n)$ rounds.*

Proof. We present an algorithm that is able to compute an $O(\Delta^2)$ coloring of a given graph G, where Δ is the maximum degree of G, in $O(\log^* n)$ rounds. By simulating such an algorithm on G^k, the k-th power of G, which has maximum degree Δ^k, we obtain the claimed result. Note that the running time is also asymptotically the same, since k is a constant.

The algorithm works as follows. At the beginning, each edge is oriented arbitrarily. Then, each node marks its incident outgoing edges with different numbers from $\{1, \dots, \Delta\}$. In this way, we decomposed our graph G into Δ edge-disjoint directed subgraphs $G_1, \dots, G_\Delta$,

where each G_i is the graph induced by edges marked i. Also, notice that by construction, for each i, each node in G_i has at most a single outgoing edge, and hence each G_i is a directed pseudoforest.

Assume we can color each directed pseudoforest with 3 colors in $O(\log^* n)$ rounds. Then, we can obtain a proper coloring for the nodes of G with 3^Δ colors, by letting each node construct the tuple $c(v) = (c_1(v), \ldots, c_\Delta(v))$, where $c_i(v)$ is the color of v in G_i. In fact, consider two neighboring nodes u and v connected through an edge e. Assume that e is oriented from u to v, and that u marked e with value i. Then, in G_i, u and v are neighbors, and hence they obtained different colors $c_i(u)$ and $c_i(v)$, implying that $c(u) \neq c(v)$. Once a 3^Δ-coloring is obtained, we can then spend $O(3^\Delta)$ rounds to reduce the number of colors to $O(\Delta^2)$, by using a simple greedy algorithm.

We now show that each pseudoforest can be 3-colored efficiently. Let P be an arbitrary pseudotree. At first, we can use the IDs of the nodes to produce a poly(n)-coloring of P. Then we apply 1 round of Linial's coloring algorithm [47] in order to obtain an $O(\log n)$-coloring of P. While this step of coloring is not necessary for the LOCAL algorithm, it allows us to reduce the amount of information that we will later need to transmit in the MPC algorithm. Nodes can then spend $T = O(\log^* n)$ rounds to gather the color of their successors in P at distance at most T, and it is known that, with this information, nodes can compute a proper coloring of P, by simulating $O(\log^* n)$ steps of a color reduction algorithm for directed paths [38, 23]. ◀

3.2 MPC Implementation

We now show how to convert the LOCAL algorithm into an exponentially faster low-space MPC algorithm. The LOCAL algorithm consists of two main steps: The distance-k $O(\Delta^{2k})$-coloring and the k-round algorithm. Since k and Δ are constant, the latter step is trivial, and the former step can be computed efficiently using graph exponentiation, where nodes keep track of the IDs of the two outermost nodes, and the colors of all nodes in between. Lemma 9 of the following paragraph proves the former step, completing the proof for Theorem 5. Component-stability and compatibility with disconnected components follows directly from the fact that all arguments are local, i.e., nodes in separate components never communicate, and that the runtime depends only on N.

Distance-k Coloring. We show that the initial distance-k coloring can be computed in $O(\log\log^* n)$ low-space MPC rounds, while respecting linear global memory. First, we observe that using the standard graph exponentiation technique, we can compute the kth power of a graph; for constant k, the memory overhead is only a constant. Then, we will apply techniques similar to the ones used in the LOCAL model in Lemma 6.

▶ **Observation 7.** *For an input graph G with n nodes, m edges, and maximum degree Δ, the power graph G^k can be computed deterministically in $O(\log k)$ low-space MPC rounds with $O(\Delta^k)$ words of local and $O(m + n \cdot \Delta^k)$ words of global memory, as long as $\Delta^k < n^\delta$.*

▶ **Observation 8.** *Every k-round LOCAL algorithm can be simulated in $O(\log k)$ low-space MPC rounds with $O(\Delta^k)$ words of local and $O(m+n\cdot\Delta^k)$ words of global memory, as long as $\Delta^k < n^\delta$. If the LOCAL algorithm is deterministic, then the MPC algorithm is deterministic as well.*

Proof. Using Observation 7, we can collect the k-hop neighborhood of each node and hence, simulate a k-round LOCAL algorithm in an additional $O(1)$ low-space MPC rounds. Observe that this also holds for general graphs. ◀

▶ **Lemma 9.** *The distance-k $O(\Delta^{2k})$-coloring problem on general graphs can be solved in the low-space MPC model with a $O(\log\log^* n + \log k)$-time deterministic algorithm, as long as $\Delta^k < n^\delta$. The algorithm requires $O(\Delta^k)$ words of local and $O(m + n \cdot \Delta^k)$ words of global memory. If k and Δ are constants, the runtime reduces to $O(\log\log^* n)$ and we require $O(1)$ words of local and $O(m+n)$ words of global memory.*

Proof. Using Observation 7, we can first compute G^k in $O(\log k)$ rounds, and operate on G^k instead of the input graph G henceforth. The application of Observation 7 requires $O(\Delta^k)$ words of local memory and $O(m + n \cdot \Delta^k)$ words of global memory. Then, similarly to Lemma 6, we can reduce the coloring problem to $O(1)$-coloring of directed pseudoforests that are initially colored with $O(\log n)$ colors.

Next, our goal is to use the graph exponentiation technique such that each node can collect the topology and the colors of its $O(\log^* n)$ successors in its pseudoforest in $O(\log\log^* n)$ time. Here, we have to take care of the subtle detail that the *color* of a successor is not enough to determine the machine on which this successor lies. Suppose that each node is initially labeled with its $O(\log\log n)$-bit color and its $O(\log n)$-bit identifier that encodes both the identity (color) of the node and the machine containing the node. Then, in round 1, each node knows the identifier and the color of its successor. For an inductive argument, suppose that each node u knows the identifier the successor v_i in distance i and the vector of colors of all nodes in between u and v_i, on the directed path from u to v_i. Then, in $O(1)$ MPC rounds, u can learn the identifier of the $2i$:th successor v_{2i} and the colors of all nodes between u and v_{2i}. After learning the identifier of v_{2i}, node u can forget about the identifier of v_i and hence, u only keeps track of one identifier. By induction, node u learns the colors of its $O(\log^* n)$ successors in $O(\log\log^* n)$ MPC rounds.

Using the vector of colors of the successors, in $O(1)$ MPC rounds, each node can simulate the $O(\log^* n)$-time LOCAL algorithm to obtain an $O(\Delta^{2k})$-coloring. This requires $O(\log^* n \cdot \log\log n + \log n) = O(\log n)$ *bits* of memory per node per pseudoforest that the node belongs to, counting the colors of the successors and the identifier of the furthest successor. Altogether, this results in a global memory requirement of $O(n \log n \cdot \Delta^k)$ bits which fits $O(n \cdot \Delta^k)$ words. ◀

4 The High Regime

In this section, we will prove that all solvable LCL problems on forests, i.e., *all LCL problems that have a correct solution on every forest*, can be solved deterministically in $O(\log n)$ time in the low-space MPC model using $O(m+n)$ words of global memory. Our proof is constructive: we explicitly provide, for any solvable LCL, an algorithm that has a runtime of $O(\log n)$. In fact, our construction can be used to find an $O(\log n)$-time algorithm *even for unsolvable LCLs*, with the guarantee that on any instance that admits a correct solution the given output will be correct (while the algorithm detects it if no solution exists). We show the following theorem.

▶ **Theorem 10.** *For any solvable LCL problem Π on a forest, there is an $O(\log n)$-time deterministic low-space MPC algorithm that is component-stable and uses $O(m+n)$ words of global memory.*

In the full version, we provide a method to solve any LCL on forests if we can solve it on trees. Hence, we can restrict attention to trees.

4.1 High-level Overview of the Algorithm and Its Analysis

Consider an arbitrary solvable LCL problem Π on trees. In the following, we will give a slightly simplified view of the algorithm we will use to solve Π in $O(\log n)$ time. First, we root the input tree by using the $O(\log n)$-round rooting algorithm described in the full version. Then, on a high level, the rest of the algorithm proceeds in 2 phases.

In the first phase, which we will refer to as the *leaves-to-root phase*, roughly speaking, the goal is to compute, for a substantial number of edges $e = (u, v)$, the set of output labels that can be output at half-edge (v, e) such that the label choice can be extended to a (locally) correct solution in the subtree hanging from v via e. This is done in an iterative manner, proceeding from the leaves towards the root. When, at last, the root has computed this set of output labels for each incident half-edge, it can, on each such half-edge, select an output label from the computed set such that the obtained node configuration is contained in the node constraint of Π and the input-output constraints of Π (given by the function g in the definition of Π) are satisfied. Such a selection must exist due to the fact that Π has a correct solution on the considered instance. We refer to these sets as the *completability information*.

The second phase, which we will refer to as the *root-to-leaves phase*, consists of completing the solution from the root downwards, by iteratively propagating the selected solution further towards the leaves. With the same argumentation as at the root, certain nodes v can select an output label at the half-edge leading to its parent and output labels from the sets computed on its incident half-edges leading to its children such that the obtained node configuration is contained in the node constraint of Π, the obtained edge configuration on the edge from v to its parent is contained in the edge constraint of Π, and the input-output constraints of Π are satisfied. The fact that the selected labels come from the sets computed in the first phase ensures that after each choice the current partial solution is part of a correct global solution. While this outline sounds simple, there are a number of intricate challenges to make the mentioned ideas work in $O(\log n)$ rounds while staying within the memory bound of $O(m + n)$.

Unfortunately, if the depth of the input tree is $\omega(\log n)$ the outlined approach has $\omega(\log n)$ steps and running them sequentially is insufficient for an $O(\log n)$-time algorithm. In order to mitigate this issue, we will not only process the leaves of the remaining unprocessed tree in each iteration, but also the nodes of degree 2, inspired by the rake-and-compress decomposition by Miller and Reif [49] which guarantees that after $O(\log n)$ iterations of removing all degree-1 and degree-2 nodes all nodes have been removed. The advantage of degree-2 nodes over higher-degree nodes w.r.t. storing completability information (as in the above outline) is that they form paths, which by definition only have two endpoints; the idea, when processing such a path, is to simply store in the two endpoints the information for which pairs of labels at the two half-edges at the ends of the path there exists a correct completion of the solution inside the path. This allows to naturally add processing degree-2 nodes to the leaves-to-root phase, while for the root-to-leaves phase, the information stored at the endpoints s, t of a path essentially allows us to start extending the current partial solution on the path itself (and thereafter on the subtrees *hanging* from nodes on the path) one step after the output labels at s and t are selected. Note that the degrees of nodes change throughout the process due to the removal of nodes and hence new nodes might become degree-2 nodes after every step of the algorithm.

Unfortunately, there are further challenges in obtaining an $O(\log n)$ runtime. In the leaves-to-root phase, even when using graph exponentiation, processing a path of degree-2 nodes of length L involves coordination between its endpoints and takes $\Omega(\log L)$ time, whereas the $O(\log n)$ time guarantee of the rake-and-compress technique crucially relies on

the fact that each iteration (optimally, an iteration would remove all leaves and all degree-2 nodes) can be performed in constant time. Hence, essentially, we will only perform one step of graph exponentiation on paths in each iteration. Here, a new obstacle arises: before the graph exponentiation is finished, new nodes (that just became degree-2 nodes due to all except one of their remaining children being conclusively processed in the most recent iteration) might join the path. Nevertheless, we will show that this process still terminates in logarithmic time by designing a fine-tuned potential function that is inspired by the idea of counting how many nodes from certain groups of degree-2 nodes are contained in any fixed "pointer chain" from some leaf to the root.

Another issue is that we have to be able to store the completability information that we compute in the leaves-to-root phase until we use it (again) in the root-to-leaves phase. Recall that the graph exponentiation technique adds new edges/pointers. Even on paths their number can be up to logarithmic in n per node (even on average), yielding a logarithmic overhead in global memory.

In order to remedy this problem, we perform preprocessing before the leaves-to-root phase, and, as a result thereof, postprocessing after the root-to-leaves phase. The preprocessing can be thought of as a more memory-efficient (hence relatively slower) version of (a few iterations in) the leaves-to-root phase. It differs by processing the degree-2 nodes, i.e., paths, in a way that guarantees that the number of new edges introduced by the graph exponentiation (which we should rather call pointer forwarding at this point) on each path in each iteration is only a constant fraction of the length of the respective path. This is achieved by finding, in each iteration, a maximal independent set (MIS) on each path, letting only MIS nodes forward pointers, and removing the MIS nodes afterwards. The preprocessing runs for $\Theta(\log\log n)$ iterations, and computing an MIS on paths in each of them takes $O(\log^* N)$ time, where N is the size of the ID space. Note that due to the removal of vertices and the way we treat paths, new paths can appear in each iteration and we need to pay the $O(\log^* N)$ runtime in each iteration, yielding a runtime of $O(\log\log n \cdot \log^* N)$ for the preprocessing, which is much less than the target runtime of $O(\log n)$ rounds.

We will show that the number of remaining nodes is $O(n/\log n)$ after the preprocessing. This property ensures that the memory overhead of $O(\log n)$ edges per node introduced in the leaves-to-root phase does not exceed the desired global memory of $O(m+n)$ words. The postprocessing runs for $\Theta(\log\log n)$ iterations and is conceptually very similar to the preprocessing. We simply iteratively extend the partial solution (computed so far) on the edges that were processed during preprocessing, analogous to the approach in the root-to-leaves phase. Lastly, we also have to ensure that the local memory restrictions of low-space MPC are not exceeded.

References

1 Noga Alon, László Babai, and Alon Itai. A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem. *Journal of Algorithms*, pages 567–583, 1986. doi:10.1016/0196-6774(86)90019-2.

2 Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel algorithms for geometric graph problems. In *Proceedings of the Symposium on Theory of Computing (STOC)*, pages 574–583, 2014. doi:10.1145/2591796.2591805.

3 Alkida Balliu, Sebastian Brandt, Yuval Efron, Juho Hirvonen, Yannic Maus, Dennis Olivetti, and Jukka Suomela. Classification of Distributed Binary Labeling Problems. In *DISC*, pages 17:1–17:17, 2020. doi:10.1145/3382734.3405703.

4. Alkida Balliu, Sebastian Brandt, Juho Hirvonen, Dennis Olivetti, Mikaël Rabie, and Jukka Suomela. Lower Bounds for Maximal Matchings and Maximal Independent Sets. In *the Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 481–497, 2019. doi:10.1109/FOCS.2019.00037.
5. Alkida Balliu, Sebastian Brandt, Fabian Kuhn, and Dennis Olivetti. Distributed Δ-Coloring Plays Hide-and-Seek. In *Proceedings of the Symposium on Theory of Computing (STOC)*, 2022. arXiv:2110.00643.
6. Alkida Balliu, Sebastian Brandt, and Dennis Olivetti. Distributed Lower Bounds for Ruling Sets. In *the Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 365–376, 2020. doi:10.1109/FOCS46700.2020.00042.
7. Alkida Balliu, Sebastian Brandt, Dennis Olivetti, Jan Studeny, Jukka Suomela, and Aleksandr Tereshchenko. Locally Checkable Problems in Rooted Trees. In *the Proceedings of the International Symposium on Principles of Distributed Computing (PODC)*, pages 263–272, 2021. doi:10.1145/3465084.3467934.
8. Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and J. Suomela. Almost Global Problems in the LOCAL Model. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 9:1–9:16, 2018. doi:10.4230/LIPIcs.DISC.2018.9.
9. Alkida Balliu, Keren Censor-Hillel, Yannic Maus, Dennis Olivetti, and Jukka Suomela. Locally Checkable Labelings with Small Messages. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 8:1–8:18, 2021. doi:10.4230/LIPIcs.DISC.2021.8.
10. Alkida Balliu, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Dennis Olivetti, and Jukka Suomela. New Classes of Distributed Time Complexity. In *Proceedings of the Symposium on Theory of Computing (STOC)*, pages 1307–1318, 2018. doi:10.1145/3188745.3188860.
11. Alkida Balliu, Juho Hirvonen, Dennis Olivetti, and Jukka Suomela. Hardness of Minimal Symmetry Breaking in Distributed Computing. In *the Proceedings of the International Symposium on Principles of Distributed Computing (PODC)*, pages 369–378, 2019. doi:10.1145/3293611.3331605.
12. Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *Journal of the ACM (JACM)*, 64(6):40, 2017. doi:10.1145/3125644.
13. Soheil Behnezhad, Sebastian Brandt, Mahsa Derakhshan, Manuela Fischer, MohammadTaghi Hajiaghayi, Richard M. Karp, and Jara Uitto. Massively Parallel Computation of Matching and MIS in Sparse Graphs. In *the Proceedings of the International Symposium on Principles of Distributed Computing (PODC)*, pages 481–490, 2019. doi:10.1145/3293611.3331609.
14. Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Łącki, and Vahab Mirrokni. Near-Optimal Massively Parallel Graph Connectivity. In *the Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1615–1636, 2020. doi:10.1109/FOCS.2019.00095.
15. Sebastian Brandt, Manuela Fischer, and Jara Uitto. Breaking the Linear-Memory Barrier in MPC: Fast MIS on Trees with Strongly Sublinear Memory. In *the Proceedings of the International Colloquium on Structural Information and Communication Complexity*, pages 124–138, 2019. doi:10.1007/978-3-030-24922-9_9.
16. Sebastian Brandt, Orr Fischer, Juho Hirvonen, Barbara Keller, Tuomo Lempiäinen, Joel Rybicki, Jukka Suomela, and Jara Uitto. A Lower Bound for the Distributed Lovász Local Lemma. In *Proceedings of the Symposium on Theory of Computing (STOC)*, pages 479–488. ACM Press, 2016. doi:10.1145/2897518.2897570.
17. Sebastian Brandt, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Patric R.J. Östergård, Christopher Purcell, Joel Rybicki, Jukka Suomela, and Przemysław Uznański. LCL Problems on Grids. In *the Proceedings of the International Symposium on Principles of Distributed Computing (PODC)*, pages 101–110, 2017. doi:10.1145/3087801.3087833.
18. Yi-Jun Chang. The Complexity Landscape of Distributed Locally Checkable Problems on Trees. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 18:1–18:17, 2020. doi:10.4230/LIPIcs.DISC.2020.18.

19 Yi-Jun Chang, Manuela Fischer, Mohsen Ghaffari, Jara Uitto, and Yufan Zheng. The Complexity of $(\Delta+1)$ Coloring in Congested Clique, Massively Parallel Computation, and Centralized Local Computation. In *the Proceedings of the International Symposium on Principles of Distributed Computing (PODC)*, pages 471–480, 2019.

20 Yi-Jun Chang, Qizheng He, Wenzheng Li, Seth Pettie, and Jara Uitto. Distributed Edge Coloring and a Special Case of the Constructive Lovász Local Lemma. *ACM Transactions on Algorithms (TALG)*, pages 1–51, 2019. doi:10.1145/3365004.

21 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An Exponential Separation between Randomized and Deterministic Complexity in the LOCAL Model. *SIAM Journal on Computing*, 48(1):122–143, 2019. doi:10.1137/17M1117537.

22 Yi-Jun Chang and Seth Pettie. A Time Hierarchy Theorem for the LOCAL Model. *SIAM Journal of Computing*, 48(1):33–69, 2019. doi:10.1137/17M1157957.

23 Richard Cole and Uzi Vishkin. Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking. *Inf. Control.*, 70(1):32–53, 1986. doi:10.1016/S0019-9958(86)80023-7.

24 Sam Coy and Artur Czumaj. Deterministic massively parallel connectivity. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2022, 2022. doi:10.1145/3519935.3520055.

25 Artur Czumaj, Peter Davies, and Merav Parter. Graph Sparsification for Derandomizing Massively Parallel Computation with Low Space. In *Proceedings of the Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 175–185, 2020. doi:10.1145/3350755.3400282.

26 Artur Czumaj, Peter Davies, and Merav Parter. Component Stability in Low-Space Massively Parallel Computation. In *the Proceedings of the International Symposium on Principles of Distributed Computing (PODC)*, pages 481–491, 2021. doi:10.1145/3465084.3467903.

27 Artur Czumaj, Peter Davies, and Merav Parter. Improved Deterministic $(\Delta+1)$ Coloring in Low-Space MPC. In *the Proceedings of the International Symposium on Principles of Distributed Computing (PODC)*, pages 469–479, 2021. doi:10.1145/3465084.3467937.

28 Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Communications of the ACM*, pages 107–113, 2008.

29 Michal Dory, Orr Fischer, Seri Khoury, and Dean Leitersdorf. Constant-Round Spanners and Shortest Paths in Congested Clique and MPC. In *the Proceedings of the International Symposium on Principles of Distributed Computing (PODC)*, pages 223–233, 2021. doi:10.1145/3465084.3467928.

30 Manuela Fischer and Mohsen Ghaffari. Sublogarithmic Distributed Algorithms for Lovász Local Lemma, and the Complexity Hierarchy. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 18:1–18:16, 2017. doi:10.4230/LIPIcs.DISC.2017.18.

31 Mohsen Ghaffari. An Improved Distributed Algorithm for Maximal Independent Set. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 270–277, 2016. doi:10.1137/1.9781611974331.ch20.

32 Mohsen Ghaffari, Christoph Grunau, and Ce Jin. Improved MPC Algorithms for MIS, Matching, and Coloring on Trees and Beyond. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 34:1–34:18, 2020. doi:10.4230/LIPIcs.DISC.2020.34.

33 Mohsen Ghaffari, Christoph Grunau, and Václav Rozhon. Improved Deterministic Network Decomposition. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2904–2923, 2021. doi:10.1137/1.9781611976465.173.

34 Mohsen Ghaffari, David G. Harris, and Fabian Kuhn. On Derandomizing Local Distributed Algorithms. In *the Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 662–673, 2018. doi:10.1109/FOCS.2018.00069.

35 Mohsen Ghaffari, Fabian Kuhn, and Jara Uitto. Conditional Hardness Results for Massively Parallel Computation from Distributed Lower Bounds. In *the Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1650–1663, 2019. doi:10.1109/FOCS.2019.00097.

36 Mohsen Ghaffari and Hsin-Hao Su. Distributed Degree Splitting, Edge Coloring, and Orientations. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2505–2523, 2017. `doi:10.1137/1.9781611974782.166`.

37 Mohsen Ghaffari and Jara Uitto. Sparsifying Distributed Algorithms with Ramifications in Massively Parallel Computation and Centralized Local Computation. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1636–1653, 2019. `doi:10.1137/1.9781611975482.99`.

38 Andrew V. Goldberg, Serge A. Plotkin, and Gregory E. Shannon. Parallel symmetry-breaking in sparse graphs. *SIAM J. Discret. Math.*, 1(4):434–446, 1988. `doi:10.1137/0401044`.

39 Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, Searching, and Simulation in the Mapreduce Framework. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 374–383, 2011. `doi:10.1007/978-3-642-25591-5_39`.

40 Christoph Grunau, Václav Rozhon, and Sebastian Brandt. The landscape of distributed complexities on trees and beyond. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*, pages 37–47. ACM, 2022. `doi:10.1145/3519270.3538452`.

41 Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *SIGOPS Operating Systems Review*, pages 59–72, 2007. `doi:10.1145/1272996.1273005`.

42 Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A Model of Computation for MapReduce. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 938–948, 2010. `doi:10.1137/1.9781611973075.76`.

43 Raimondas Kiveris, Silvio Lattanzi, Vahab Mirrokni, Vibhor Rastogi, and Sergei Vassilvitskii. Connected Components in MapReduce and Beyond. In *ACM Symposium on Cloud Computing*, pages 18:1–18:13, 2014. `doi:10.1145/2670979.2670997`.

44 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: Lower and upper bounds. *Journal of the ACM (JACM)*, 63:17:1–17:44, 2016. `doi:10.1145/2742012`.

45 Christoph Lenzen and Roger Wattenhofer. Brief Announcement: Exponential Speed-Up of Local Algorithms Using Non-Local Communication. In *the Proceedings of the International Symposium on Principles of Distributed Computing (PODC)*, pages 295–296, 2010. `doi:10.1145/1835698.1835772`.

46 Nathan Linial. Distributive Graph Algorithms - Global Solutions from Local Data. In *the Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 331–335, 1987. `doi:10.1109/SFCS.1987.20`.

47 Nathan Linial. Locality in Distributed Graph Algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992. `doi:10.1137/0221015`.

48 Michael Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. In *Proceedings of the Symposium on Theory of Computing (STOC)*, pages 1–10, 1985. `doi:10.1145/22145.22146`.

49 Gary L. Miller and John H. Reif. Parallel Tree Contraction Part 1: Fundamentals. *Adv. Comput. Res.*, 5:47–72, 1989.

50 Moni Naor and Larry Stockmeyer. What Can Be Computed Locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995. `doi:10.1137/S0097539793254571`.

51 Alessandro Panconesi and Romeo Rizzi. Some Simple Distributed Algorithms for Sparse Networks. *Distributed Computing*, 14(2):97–100, 2001. `doi:10.1007/PL00008932`.

52 David Peleg. *Distributed Computing: A Locality-Sensitive Approach.* Society for Industrial and Applied Mathematics, 2000. `doi:10.1137/1.9780898719772`.

53 Tim Roughgarden, Sergei Vassilvitskii, and Joshua R. Wang. Shuffles and Circuits (On Lower Bounds for Modern Parallel Computation). *J. ACM*, pages 1–12, 2018. `doi:10.1145/3232536`.

54 Václav Rozhon and Mohsen Ghaffari. Polylogarithmic-Time Deterministic Network Decomposition and Distributed Derandomization. In *Proceedings of the Symposium on Theory of Computing (STOC)*, pages 350–363, 2020. `doi:10.1145/3357713.3384298`.

55 Tom White. *Hadoop: The Definitive Guide.* O'Reilly Media, Inc., 2012. doi:10.5555/1717298.
56 Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2010. doi:10.5555/1863103.1863113.

A Component-stability

The notion of a *component-stable* MPC algorithm has been introduced in [35] in the context of lifting distributed lower bounds to the MPC setting. It was later revised by Czumaj, Davies and Parter [26] and argued to be made more robust.

▶ **Definition 11** (Component-stability, [35]). *An MPC algorithm is component-stable if the outputs of nodes in different connected components are independent. Formally, assume that for a graph G, $\mathcal{D}_G$ denotes the initial distribution of the edges of G among the M machines and the assignment of unique IDs to the nodes of G. For a subgraph H of G let $\mathcal{D}_H$ be defined as $\mathcal{D}_G$ restricted to the nodes and edges of H. Let H_v be the connected component of node v. An MPC algorithm $\mathcal{A}$ is called component-stable if for each node $v \in V$, the output of v depends (deterministically) on the node v itself, the initial distribution and ID assignment $\mathcal{D}_{H_v}$ of the connected component H_v of v, and on the shared randomness $\mathcal{S}_M$.*

In their revised definition, [26] assume the setting where all input graphs are legal.

▶ **Definition 12** (Legal graph). *A graph G is called legal if it is equipped with functions ID, name: $V(G) \to [\text{poly}(n)]$ providing nodes with IDs and names, such that all names are fully unique and all IDs are unique in every connected component.*

▶ **Definition 13** (Component-stability (revised), [26]). *A randomized MPC algorithm A_{MPC} is component-stable if its output at any node v is entirely, deterministically, dependent on the topology and IDs (but independent of names) of v's connected component (which we will denote $CC(v)$), v itself, the exact number of nodes n and maximum degree Δ in the* entire *input graph, and the input random seed $\mathcal{S}$. That is, the output of A_{MPC} at v can be expressed as a deterministic function $A_{MPC}(CC(v), v, n, \Delta, \mathcal{S})$. A deterministic MPC algorithm A_{MPC} is component-stable under the same definition, but omitting dependency on the random seed $\mathcal{S}$.*

As opposed to [35], [26] allow the output of component-stable algorithms to depend on the total number of nodes in the graph and the maximum degree of the graph. Additionally, they assume the following setting: all input graphs are *legal* (see Definition 12), i.e., all nodes have an ID that is unique in every connected component, and a *name* that is unique across the whole input graph. Assuming the above setting, the output of a component-stable algorithm is allowed to depend on the IDs of all nodes in the same components, but not the names.

In our work, we adopt the revised definition of component-stability [26]. In all of our algorithms, nodes from different components only communicate in order to maintain a certain global synchrony. This synchrony influences *when* certain steps are executed and hence the *execution* of our algorithms. However, the output at each node is not influenced by the global communication.

Theorem 2 shows that the lower bounds for component-stable algorithms can be beaten for a large class of problems on trees and forests even with optimal memory. The long term effect of the term component-stable in this setting is unclear, but it provides room for many interesting open questions. One interesting aspect would be to see under which circumstances one can obtain algorithms with stronger component dependent guarantees, e.g., one may

want to develop algorithms for which not just the output of a node, but also the time until it has computed its output can only depend on the size of its component. Our algorithms do not meet this stronger definition. Besides an ID space dependence our algorithms have the following runtime behaviour. In the low and mid regime the time until we know the output of a node depends on the number of nodes in the largest connected component. In the high regime this time depends on the number of nodes in the whole graph. Going from trees to forests in the high regime relies on the recent beautiful (deterministic) connected components algorithm by Czumaj and Coy [24, 14].

Holistic Verification of Blockchain Consensus

Nathalie Bertrand
INRIA Rennes, France

Vincent Gramoli
University of Sydney, Australia
Redbelly Network, Sydney, Australia

Igor Konnov
Informal Systems, Wien, Austria

Marijana Lazić
TU München, Germany

Pierre Tholoniat
Columbia University, New York, NY, USA

Josef Widder
Informal Systems, Wien, Austria

Abstract

Blockchain has recently attracted the attention of the industry due, in part, to its ability to automate asset transfers. It requires distributed participants to reach a consensus on a block despite the presence of malicious (a.k.a. Byzantine) participants. Malicious participants exploit regularly weaknesses of these blockchain consensus algorithms, with sometimes devastating consequences. In fact, these weaknesses are quite common and are well illustrated by the flaws in various blockchain consensus algorithms [67]. Paradoxically, until now, no blockchain consensus has been holistically verified.

In this paper, we remedy this paradox by model checking for the first time a blockchain consensus used in industry. We propose a holistic approach to verify the consensus algorithm of the Red Belly Blockchain [20], for any number n of processes and any number $f < n/3$ of Byzantine processes. We decompose directly the algorithm pseudocode in two parts – an inner broadcast algorithm and an outer decision algorithm – each modelled as a threshold automaton [37], and we formalize their expected properties in linear-time temporal logic. We then automatically check the inner broadcasting algorithm, under a carefully identified fairness assumption. For the verification of the outer algorithm, we simplify the model of the inner algorithm by relying on its proven properties. Doing so, we formally verify, for any parameter, not only the safety properties of the Red Belly Blockchain consensus but also its liveness in less than 70 seconds.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms; Theory of computation → Logic and verification

Keywords and phrases Model checking, automata, logic, byzantine failure

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.10

Funding This research is supported under Australian Research Council Future Fellowship funding scheme (project number 180100496) entitled "The Red Belly Blockchain: A Scalable Blockchain for Internet of Things", Interchain Foundation (Switzerland), and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement No 787367 (PaVeS).

© Nathalie Bertrand, Vincent Gramoli, Igor Konnov, Marijana Lazić, Pierre Tholoniat, and Josef Widder;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

1.1 Context

As blockchains require a distributed set of machines to agree on a unique block of transactions to be appended to the chain, attackers naturally try to exploit consensus vulnerabilities: they force participants to disagree so that they wrongly believe that two conflicting transactions are legitimate, leading to what is known as a *double spending.* In 2014, malicious participants managed to exploit Bitcoin consensus vulnerabilities to steal $83,000 through a network attack. In August 2021, 570,000 transactions were reverted in a more recent version of Bitcoin, Bitcoin SV, by forcing its blockchain consensus protocol to violate its safety property (i.e., agreement). With 3 attacks on the same blockchain within 4 months, thefts are becoming commonplace.[1] Unsurprisingly, various bugs in specifications and in proofs of blockchain consensus protocols appear in the literature [1, 65]. This is illustrated by the flaws in the consensus algorithms now used in in-production blockchains [67]. The crux of the problem is that reasoning about distributed executions of blockchain consensus protocols is hard due to several sources of non-determinism, and in particular asynchrony and faults. As a result, formally verifying that a blockchain consensus protocol is safe and live is key to mitigate financial losses.

Recent progress in mechanical proofs represent the first steps towards verifying blockchain consensus. For instance, parameterized model checking aims at verifying algorithms for an arbitrary number n of processes [11] that is unknown at design time. In some contexts, it reduces the model checking for any fault number f and its upper bound t to bounded model checking questions [30]. The threshold automaton (TA) framework for communication-closed algorithms [37, 7] targets algorithms with thresholds in guards such as "number of messages from distinct processes exceeds $2t + 1$", and in the resilience condition, typically of the form $n > 3t$. The parameterized model checking of threshold automata builds upon a reduction [27, 43] that reorders steps of asynchronous executions to obtain simpler executions, which are equivalent to the original executions with respect to safety and liveness properties. Such a technique has recently proved instrumental in verifying fully asynchronous parts of consensus algorithms, like broadcast algorithms [37].

Due to the famous unfeasability of deterministic consensus in asynchronous setting [29], this promising method was not applied to proving deterministic consensus algorithms correct[2]. In fact, the aforementioned reduction technique cannot apply to partial synchrony [25]: moving the message reception step to a later point in the execution might violate an assumed message delay. Yet, these delays are important as typical partially synchronous consensus algorithms feature timers to catch up with the unknown bound on the delay to receive a message. Most known verification techniques therefore target either synchronous (lock-step) or asynchronous semantics. In addition, partially synchronous consensus algorithms generally rely on a coordinator process that helps other processes converge and whose identifier rotates across rounds. Some efforts have been devoted to proving the termination of partially synchronous consensus algorithms, like Paxos, assuming synchrony [31]. The drawback is that such algorithms aim at tolerating non-synchronous periods before reaching a global stabilization time (GST) after which they terminate. Proving that such an algorithm terminates under synchrony does not show that the algorithm would also terminate if processes reached GST at

[1] https://cointelegraph.com/news/bitcoin-sv-rocked-by-three-51-attacks-in-as-many-months
[2] Here deterministic means that randomization is forbidden. However, the environment (*e.g.*, communication delays, scheduler) introduces non-determinism in the algorithm execution.

different points of their execution. Instead, one would also need to show that correct processes can catch up in the same round. This would, in turn, require proving the correctness of a synchronizer algorithm [25].

Verifying consensus is even more subtle when processes are Byzantine as they can execute arbitrary steps, changing their local state and the values they share. One needs to reason about executions with all possible scenarios resulting from arbitrary behaviors, multiplying the already large number of interleaved executions. This is probably the reason why, to our knowledge, blockchain consensus algorithms have never been holistically verified. Despite recent efforts towards proving consensus algorithms automatically, these were limited to proving safety properties [3, 9], to checking proofs [46, 45], to synthesizing parameterized distributed algorithms [66, 26, 48, 49], to deductively verifying implementations [21] or to proving algorithms with fixed parameters [34]. Without a holistic approach, the verification of parts of a protocol does not imply that the protocol is verified.

1.2 Contributions

In this paper, we verify holistically the safety and liveness properties of the Byzantine consensus protocol used in the Red Belly Blockchain system [20], a scalable blockchain used in industry. Our approach is *holistic* because it starts from the pseudocode of the distributed algorithm as typically presented in the distributed computing literature, models this pseudocode and its components into disambiguated threshold automata (TAs), model checks both the safety and liveness properties of these components expressed in linear temporal logic (LTL) formulae, and for any parameters n and $f < n/3$. The advantage is that the formally verified algorithm matches the pseudocode and no user-defined invariants or proofs need to be checked, which drastically reduces the risks of human errors.

1. We formally verify a Byzantine consensus algorithm [19] used for e-voting [14], for accountability [18] and for blockchains [20]. This consensus algorithm now runs in the network of the Red Belly Blockchain [20] maintained by the Redbelly Network company. It executes in asynchronous rounds that broadcast binary values and compares the delivered values to the parity of the round to decide. To model check the algorithm holistically, we replace the partial synchrony assumption by a fairness assumption. Interestingly, our fairness assumption only requires that in any infinite sequence of rounds, there exists a round where, at all correct processes, a broadcast instance delivers the same binary value, or bit, first.
2. We exploit the modularity of distributed algorithms in parameterized model checking. We first model the consensus algorithm into two simpler algorithms modeled as threshold automata (TAs): (i) an inner broadcast TA modeling a binary value variant of the reliable broadcast [50] and (ii) an outer decision TA modeling a round-based execution that inspects the delivered messages [19] to decide. We express the guarantees of the inner broadcast primitive as temporal logic properties that we automatically verify and we replace the inner TA in the global TA by a gadget TA that captures the proven temporal specification. We automatically verify the global TA with model checking.
3. We show the practicality of our verification technique by running the parameterized model checker ByMC [37] for any number n of processes and any arbitrary number $f < n/3$ of Byzantine processes. We compare the execution times when model checking the naive TA encoding the consensus algorithm and when model checking both the inner TA encoding the broadcast algorithm and then the outer TA. We demonstrate empirically that, although a parallel execution of ByMC on 64 cores could not prove the safety of the naive TA within 3 days, it proves both the liveness and safety of the simplified TA in about 70 seconds.

1.3 Outline

In Section 2 we introduce our preliminary definitions, in Section 3 we model our binary value broadcast algorithm pseudocode into a corresponding threshold automaton, in Section 4 we explain how the formal verification of the properties of the broadcast algorithm helps us model check the consensus algorithm and in Section 5 we verify the consensus algorithm. In Section 6 we present the experimental results of the model checker. In Section 7, we present the related work and in Section 8, we conclude. In the Appendix we explain the multiple-round TA to one-round TA reduction (A), provide examples related to fairness (B), missing proofs (C and E) and detailed specifications (D and F).

2 Preliminaries

The consensus algorithm runs over n asynchronous sequential processes from the set $\Pi = \{p_1, \ldots, p_n\}$. The processes communicate by exchanging messages through an asynchronous reliable fully connected point-to-point network, hence there is no bound on the delay to transfer a message but this delay is finite.

Failure model. Up to $t < n/3$ processes can exhibit a *Byzantine* behavior [56], and behave arbitrarily. We refer to $f \leq t$ as the actual number of Byzantine processes. A Byzantine process is called *faulty*, a non-faulty process is *correct*.

Algorithm semantics. The asynchronous semantics of a distributed algorithm executed by processes in Π assumes discrete time and at each point in time, exactly one process takes a step. We assume that two messages cannot be received at the same time by the same process. The global execution then consists in an interleaving of the individual steps taken by the processes. Process p_i sends a message to p_j by invoking the primitive "send $\textsc{header}(m)$ to p_j", where HEADER indicates the type of message and m its contents. Process p_i receives a message by executing the primitive "receive()". The shorthand broadcast($\textsc{header}, m$) represents "for each $p_j \in \Pi$ do send $\textsc{header}(m)$ to p_j". And the right arrow in broadcast($\textsc{header}, m$) $\rightarrow$ *messages* indicates, when specified, that "upon reception of $\textsc{header}(m)$ from process p'_j do $messages[p'_j] \leftarrow messages[p'_j] \cup \{m\}$". The process id is used as a subscript to denote that a variable is local to a process – for instance var_i is local to process p_i – and is omitted when it is clear from the context.

The verification method considered in this paper exploits the fact that the algorithms are communication-closed [27], *i.e.* only messages from the current loop iteration or *round* of a process may influence its steps. This can be implemented by tagging every message by its round number r; during round r all received messages with tag $r' < r$ are discarded and all received messages with tag $r' > r$ are stored for later.

The consensus problem. Assuming that each correct process proposes a binary value, the binary Byzantine consensus problem is for each of them to decide on a binary value in such a way that the following properties are satisfied:

1. Termination. Every correct process eventually decides on a value.
2. Agreement. No two correct processes decide on different values.
3. Validity. If all correct processes propose the same value, no other value can be decided.

Threshold automaton (TA). A *threshold automaton* [38] describes the behavior of a process in a distributed algorithm. Its nodes are *locations* representing local states, and labeled edges are *guarded rules*. Formally, it is a tuple $\langle \mathcal{L}, \mathcal{I}, \Gamma, \mathcal{P}, \mathcal{R}, RC \rangle$ where $\mathcal{L}$ is the set of locations, $\mathcal{I} \subset \mathcal{L}$ is the set of initial locations, Γ is the set of shared variables that all processes can update, $\mathcal{P}$ is the finite set of parameter variables, $\mathcal{R}$ is the set of rules, and RC is the resilience condition over $\mathbb{N}_0^{|\Pi|}$. Rules are defined as tuples $\langle from, to, \phi, \vec{u} \rangle$, where *from* (resp. *to*) describes the source (resp. destination) locations, and the rule label is $\phi \mapsto \vec{u}$. Formula ϕ is called a *threshold guard* or simply a *guard*.

1: bv-broadcast(BV, $\langle val, i \rangle$)**:**
2: broadcast(BV, $\langle val, i \rangle$)
3: **repeat:**
4: **if** (BV, $\langle v, * \rangle$) received from $(t+1)$ distinct processes and not yet re-broadcast **then**
5: broadcast(BV, $\langle v, i \rangle$)
6: **if** (BV, $\langle v, * \rangle$) received from $(2t+1)$ distinct processes **then**
7: $contestants \leftarrow contestants \cup \{v\}$

Figure 1 The pseudocode of the binary value broadcast for process p_i.

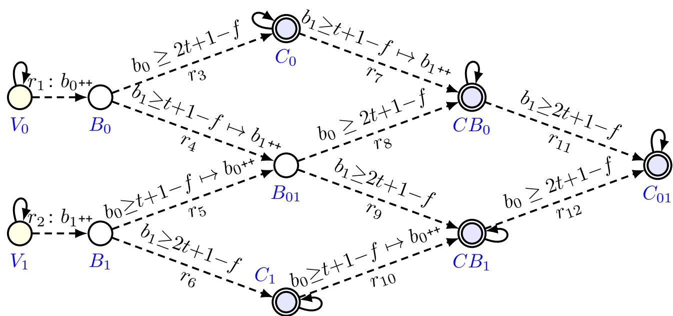


Figure 2 The threshold automaton model for the binary value broadcast.

▶ **Example 1.** As an example, Fig. 1 presents the pseudocode of the binary value broadcast and Fig. 2 its TA. (The modeling of pseudocode (Fig. 1) into TA (Fig. 2) will be described in detail in Section 3.1.) To illustrate the TA notations, note that two of the locations in $\mathcal{L} = \{V_0, V_1, B_0, B_1, B_{01}, C_0, C_1, CB_0, CB_1, C_{01}\}$ are initial: $\mathcal{I} = \{V_0, V_1\}$. Shared variables are b_0 and b_1 and can be updated by each process traversing the TA, while parameters are n, t and f and remain unchanged across the execution. The set of rules $\mathcal{R}$ consists of $\{r_i \mid 1 \leq i \leq 12\}$ together with 7 self-loops. The self- loops mimic the asynchrony between processes in the system. For example, rule r_3 is defined as $\langle B_0, C_0, b_0 \geq 2t+1-f, \vec{0} \rangle$. The resilience condition is $n > 3t \wedge t \geq f \geq 0$.

A *multi-round threshold automaton* is intuitively defined such that one round is represented by a threshold automaton, and additional so-called *round-switch rules* connect final locations with initial ones, and therefore allow processes to move from one round to the following one. We typically depict those round-switch rules as dotted arrows. Examples of such multi-round TA are depicted later in Figures 3 and 4. When it is clear from the context that there are multiple rounds, we simply call them threshold automata, and to stress that a TA does not have multiple rounds, we may call it a one-round TA.

Counter systems. The semantics of a (one-round) threshold automaton TA are given by a counter system $Sys(\mathsf{TA}) = \langle \Sigma, I, T \rangle$ where Σ is the set of all configurations among which I are the initial ones, and T is the transition relation. A configuration $\sigma \in \Sigma$ of a one-round TA captures the values of location counters (counting the number of processes at each location of $\mathcal{L}$, therefore non-negative integers), values of global variables, and parameter values. A transition $t \in T$ is *unlocked in* σ if there exists a rule $r = \langle from, to, \phi, \vec{u} \rangle \in \mathcal{R}$ such that ϕ evaluates to true in σ, and location counter of *from* is at least 1, denoted $\kappa[from] \geq 1$, showing that at least one process is currently in *from*. In this case we can execute transition t on σ by moving a process along the rule r from location *from* to location *to*, which is modeled by decrementing counter $\kappa[from]$, incrementing $\kappa[to]$, and updating global variables according to the update vector $\vec{u}$.

A counter system $Sys(\mathsf{TA})$ of a multi-round TA is defined analogously. A configuration captures the values of location counters and global variables *in each round*, and parameter values (that do not change over rounds). Then we define that a transition is *unlocked in a round* R by evaluating the guard ϕ and the counter of location *from* in the round R. The execution of the transition in σ accordingly updates $\kappa[from, R]$, $\kappa[to, R]$ and global variables of that round, while the values of these variables in other rounds stay unchanged.

Linear temporal logic notations. Following a standard model checking approach, we use formulas in linear temporal logic (LTL) [57] to formalize the desired properties of distributed algorithms. The basic elements of these formulas, called atomic propositions, are predicates over configurations related (i) to the emptiness of each location at each round and (ii) to the evaluation of threshold guards in each round. They have the following form: (i) $\kappa[L, R] \neq 0$ expresses that at least one correct process is in location L in round R, while $\kappa[L, R] = 0$ expresses the opposite (in one-round systems we just write $\kappa[L] \neq 0$ or $\kappa[L] = 0$); (ii) the evaluation of $[b_0, R]{\geq}2t{+}1{-}f$ depends on the values of the shared variable b_0 in round R and parameters t and f (in one-round systems we just write $b_0{\geq}2t{+}1{-}f$). LTL builds on propositional logic with $\Rightarrow$ for "implication", $\vee$ for 'or' and $\wedge$ for "and", and has extra temporal operators $\Diamond$ standing for "eventually" and $\Box$ for "always". LTL formulas are evaluated over infinite runs of $Sys(\mathsf{TA})$. Examples of LTL properties in a one-round system are $(BV - Just_v)$, $(BV - Obl_v)$ and $(BV - Unif_v)$ (see page 9). LTL properties in multi-round systems often have quantifiers over round variables, as for example in $(Agree_v)$ and $(Valid_v)$ (see page 13).

The tool ByMC is used to automatically verify a specific fragment of LTL on one-round systems [36, 37], which is sufficient to express safety and liveness properties of consensus [10]. Moreover, thanks to communication-closure, the verification for this fragment of temporal logic on multi-round systems reduces to one-round systems [10, Theorem 6] (see also Appendix A).

The assumption of reliable communication is modeled as follows at the TA level: if the guard of a rule is true infinitely often, then the origin location of that rule will eventually be empty. This reflects that an if branch of the pseudo-code is taken if the condition is true. This *progress assumption* is in particular crucial to prove liveness properties: in the sequel, we prepend it to the liveness properties in the TA specification.

3 The Binary Value Broadcast

To overcome the limited scalability of model checking tools, our holistic verification approach consists of decomposing a distributed algorithm into encapsulated components of pseudocode that can be modelled in threshold automata and verified in isolation to obtain a simplified threshold automaton that is amenable to automated verification.

In this section we focus on a *binary value broadcast*, or bv-broadcast for short, that will serve as the main building block of the Byzantine consensus algorithm of Section 4. In Section 3.1 we formally model the bv-broadcast algorithm pseudocode as a threshold automaton that tolerates a number f of Byzantine failures upper-bounded by t among n processes. In Section 3.2 we model the specification of bv-broadcast in LTL and verify, within 10 seconds, that it holds. In Section 3.3 we introduce the fairness of an infinite sequence of executions of bv-broadcast that will play a crucial role in verifying holistically in Section 5 the Byzantine consensus algorithm.

3.1 Modeling the binary value broadcast pseudocode into a threshold automaton

The binary value broadcast [50], or *bv-broadcast* for short, is a communication primitive guaranteeing that all binary values "bv-delivered" were "bv-broadcast" by a correct process. It is particularly useful to solve the Byzantine consensus problem with randomization [51, 15] or partial synchrony [19, 14]. As discussed before, Figures 1 and 2 in Section 2 give its pseudocode and the corresponding threshold automaton, respectively. We now explain how we model our bv-broadcast pseudocode (Fig. 1) parameterized by n and f into a threshold automaton (Fig. 2) using the synthesis methodology [42].

Pseudocode of the binary value broadcast. The bv-broadcast algorithm pseudocode (Fig. 1) aims at having at least $2t{+}1$ processes broadcasting the same binary value. Each process starts this algorithm in one of two states, depending on its input value 0 or 1. Once a correct process receives a value from $t{+}1$ distinct processes, it broadcasts it (line 4) if it did not do it already (line 4); broadcast is not Byzantine fault tolerant and just sends a message to all the other processes. Once a correct process receives a value from $2t{+}1$ distinct processes, it delivers it. Here the delivery at process p_i is modeled by adding the value to the set *contestants*, which will simplify the pseudocode of the Byzantine consensus algorithm in Section 4.1.

Threshold automaton of the binary value broadcast. To match the two initial states from which a process starts the bv-broadcast algorithm, we start the corresponding TA of Fig. 2 with two initial locations V_0 or V_1, indicating whether the (correct) process initially has value 0 or 1, resp. We can see form the pseudocode (Fig. 1) that a correct process p_i sends only two types of messages, $(\mathsf{BV}, \langle 0, i\rangle)$ and $(\mathsf{BV}, \langle 1, i\rangle)$, these trigger the corresponding receptions at other processes. We thus define in the TA (Fig. 2) two global variables b_0 and b_1, resp., to capture the number of the two types of messages sent by correct processes. Thus, for example, b_0++ models a process broadcasting message $(\mathsf{BV}, \langle 0, i\rangle)$. Because the algorithm only counts messages regardless of sender identities, we replace the messages from the pseudocode into b_0 and b_1 shared variables that are increased whenever a message is sent.

From local to global variables for model checking. While producing a formal model, extra care is needed to avoid introducing redundancies. For example, line 4 indicates that the process broadcasts value v if it received v from $t{+}1$ distinct processes. Instead of maintaining local receive variables, it is sufficient to enable a guard based on global send variables. Indeed, to remove redundant local receive variables, one can use the quantifier elimination for Presburger arithmetic [58] and obtain quantifier-free guard expressions over the shared variables that are valid inputs to ByMC [39, 35]. For more details, note that Stoilkovska

et al. [64] eliminated the quantifier over the similar receive variables in Ben-Or's consensus algorithm [8] with the SMT solver Z3 [22]. Finally, the point-to-point reliable channels ensure that p_j sends message m to p_i implies that eventually p_i receives message m from p_j. Hence shared variables b_0 and b_1 of the TA denote, respectively, the number of messages $(\mathsf{BV}, \langle 0, i\rangle)$ and $(\mathsf{BV}, \langle 1, i\rangle)$ *sent* by correct processes in the pseudocode.

Modeling arbitrary (Byzantine) behaviors in the TA. In order to model that, among the received messages, f messages could have been sent by Byzantine processes, we need to map the "if" statement of the pseudocode, comparing the number of receptions from distinct processes to t+1, to the TA guards, comparing the number b_1+f of messages sent to $t+1$. As b_1 counts the messages *sent* by correct processes and f is the number of faulty processes that can send arbitrary values, a correct process can move from B_0 to B_{01} as soon as t+1−f correct processes have sent 1, provided that f faulty processes have also sent 1. As a result, the guard of rule r_4 only evaluates over global send variables as: if more than t+1 messages of type b_1 have been *sent* by correct processes (hence the guard $b_1 \geq t$+1−f), then the shared variable b_1 is incremented, mimicking the *broadcast* of a new message of type b_1. Rule r_3 corresponds to lines 6–7 and delivers value $v = 0$ by storing it into variable *contestants* upon reception of this value from $2t+1$ distinct processes. Hence, reaching location C_0 in the TA indicates that the value 0 has been delivered. As a process might stay in this location forever, we add a self-loop with guard condition set to true.

Table 1 The locations of correct processes.

locations	V_0	V_1	B_0	B_1	B_{01}	C_0	CB_0	C_1	CB_1	C_{01}
val. broadcast	/	/	0	1	0,1	0	0,1	1	0,1	0,1
val. delivered	/	/	/	/	/	0	0	1	1	0,1

Other locations and rules. The locations of the automaton correspond to the exclusive situations for a correct process depicted in Table 1. After location C_0, a process is still able to broadcast 1 and eventually deliver 1 after that. After location B_{01}, a process is able to deliver 0 and then deliver 1, or deliver 1 first and then deliver 0, depending on the order in which the guards are satisfied. Apart from the self-loops, note that the automaton is a directed acyclic graph. Also, on every path in the graph, a shared variable is incremented only once. This reflects that in the pseudocode, a value may only be broadcast if it has not been broadcast before.

3.2 Properties of the binary value broadcast

As was previously proved by hand [50, 51], the bv-broadcast primitive satisfies four properties: BV-Justification, BV-Obligation, BV-Uniformity and BV-Termination. Here, we formalize these properties in linear temporal logic (LTL) to formally and automatically prove they hold. As we will discuss in Section 6, we verify them for any parameters n and $t < n/3$ in less than 10 seconds.

The BV-Justification property states: "If p_i is correct and $v \in contestants_i$, then v has been bv-broadcast by some correct process" where $v \in \{0, 1\}$. Alternatively, "if v is not bv-broadcast by some correct process and p_i is correct, then $v \notin contestants_i$". In the TA

from Fig. 2, $v \in \mathit{contestants}_i$ corresponds to process i being in one of the locations C_v, CB_v or C_{01}. Thus, justification can be expressed in LTL as the conjunction $BV\text{-}Just_0 \wedge BV\text{-}Just_1$ where, $BV\text{-}Just_v$ is the following formula:

$$\kappa[V_v] = 0 \;\Rightarrow\; \Box \left(\kappa[C_v] = 0 \wedge \kappa[CB_v] = 0 \wedge \kappa[C_{01}] = 0\right) \;. \qquad (BV\text{-}Just_v)$$

BV-Obligation requires that if at least $(t{+}1)$ correct processes bv-broadcast the same value v, then v is eventually added to the set $\mathit{contestants}_i$ of each correct process p_i. This can again be formalized as $BV\text{-}Obl_0 \wedge BV\text{-}Obl_1$ where $BV\text{-}Obl_v$ is the following formula:

$$\Box \Big(b_v \geq t{+}1 \;\Rightarrow\; \Diamond \big(\bigwedge_{L \in \mathrm{Locs}_v} \kappa[L] = 0\big)\Big) \;, \qquad (BV\text{-}Obl_v)$$

where $\mathrm{Locs}_v = \{V_0, V_1, B_0, B_1, B_{01}, C_{1-v}, CB_{1-v}\}$ are all the possible locations of a process i if $v \notin \mathit{contestants}_i$.

BV-Uniformity requires that if a value v is added to the set $\mathit{contestants}_i$ of a correct process p_i, then eventually $v \in \mathit{contestants}_j$ at every correct process p_j. We formalize this as $BV\text{-}Unif_0 \wedge BV\text{-}Unif_1$ where $BV\text{-}Unif_v$ is the following:

$$\Diamond \left(\kappa[C_v]{\neq}0 \;\vee\; \kappa[CB_v]{\neq}0 \;\vee\; \kappa[C_{01}]{\neq}0\right) \;\Rightarrow\; \Diamond \bigwedge_{L \in \mathrm{Locs}_v} \kappa[L]{=}0 \;, \qquad (BV\text{-}Unif_v)$$

where Locs_v is defined as in $(BV\text{-}Obl_v)$.

Finally, the BV-Termination property claims that eventually the set $\mathit{contestants}_i$ of each correct process p_i is non empty. This can be phrased as the following LTL formula $BV\text{-}Term$:

$$\Diamond \left(\kappa[V_0]{=}0 \;\wedge\; \kappa[V_1]{=}0 \;\wedge\; \kappa[B_0]{=}0 \;\wedge\; \kappa[B_1]{=}0 \;\wedge\; \kappa[B_{01}]{=}0\right) \;, \qquad (BV\text{-}Term)$$

forcing each correct process to be in one of the "final" locations $C_0, C_1, C_{01}, CB_0, CB_1$.

3.3 A fairness assumption to solve consensus

The traditional approach to establishing guarantee properties in verification is to require that all fair computations, instead of all computations, satisfy the property [4]. We thus introduce a crucial fairness assumption. In order to define it, we first define a *good execution* of the bv-broadcast with respect to binary value v as an execution:

▶ **Definition 2** (v-good bv-broadcast)**.** *A* bv-broadcast *execution is* v-good *if all its correct processes bv-deliver* v *first.*

We express this property in LTL. A bv-broadcast execution is v*-good* if no process ever visits locations C_{1-v} and CB_{1-v}:

$$\Box \;\Big(\kappa[C_{1-v}] = 0 \;\wedge\; \kappa[CB_{1-v}] = 0\Big) \;.$$

Second, we consider an infinite sequence of bv-broadcast executions, tagged with $r \in \mathbb{N}$. It is important to stress that the setting is asynchronous, that is, processes invoke bv-broadcast infinitely many times, but at their own relative speed. Thus, they do not all invoke the bv-broadcast tagged with the same number r at the same time. Nonetheless, every process invokes bv-broadcast infinitely many times and in the rth invocation its behavior depends on the messages sent in the rth invocation of other processes. Therefore, we refer to the rth execution of bv-broadcast even though the processes invoke it at different times.

▶ **Definition 3** (fairness). *An infinite sequence of* bv-broadcast *executions is* fair *if there exists an r such that the rth execution is* $(r \bmod 2)$*-good.*

For simplicity, we use the terminology *fair* bv-broadcast when the infinite sequence of bv-broadcast executions is fair. We illustrate in Appendix B a possible execution of bv-broadcast whose existence implies fairness.

4 Simplified Automaton for Byzantine Consensus

In this section we exploit the results of the first verification phase of Section 3 to simplify the threshold automaton of the Byzantine consensus algorithm. In Section 4.1 we introduce the pseudocode of the Byzantine consensus algorithm and its threshold automaton obtained with the naive modeling described in Section 3.1. In Section 4.2 we replace, in this threshold automaton, the inner bv-broadcast automaton by a smaller one obtained thanks to the bv-broadcast properties that are now verified. The verification of the resulting simplified automaton is deferred to Section 5.

4.1 The Byzantine consensus algorithm

Algorithm 1 is the DBFT Byzantine consensus algorithm [19] that relies on the fair binary value broadcast of Section 3. It is currently used in the Red Belly Blockchain, a recent blockchain that achieves unprecedented scalability [20]. More precisely, the DBFT binary consensus comes in two different variants: (i) a first variant that is safe but not live in the asynchronous setting, (ii) a second variant that is safe and live under the partial synchrony assumption. We use the first variant of it (without coordinator or timeout) here and show that it is live under our new fairness assumption. The DBFT binary consensus invokes bv-broadcast$(\cdot)$ at line 6 and uses a set *contestants* of binary values, whose scope is global, updated by the bv-broadcast (Fig. 1, line 7) and accessed by propose$(\cdot)$ (Alg. 1, line 7).

Algorithm 1 The Byzantine consensus algorithm at process p_i.

1: **Global scope variable:**
2: $\quad$ $contestants \subseteq \{0,1\}$, set of binary values, init. $\emptyset$.

3: propose(est)**:**
4: $\quad r \leftarrow 0$
5: $\quad$ **repeat:**
6: $\quad\quad$ bv-broadcast$(\textsc{est}, \langle est, i\rangle)$
7: $\quad\quad$ **wait until** $(contestants \neq \emptyset)$
8: $\quad\quad$ broadcast$(\text{AUX}, \langle contestants, i\rangle) \rightarrow favorites$
9: $\quad\quad$ **wait until** $\exists c_1, \ldots, c_{n-t} : \forall 1 \leq j \leq n-t \; favorites[c_j] \neq \emptyset \wedge (qualifiers \leftarrow \cup_{\forall 1 \leq j \leq n-t} favorites[c_j]) \subseteq contestants$
10: $\quad\quad$ **if** $qualifiers = \{v\}$ **then**
11: $\quad\quad\quad est \leftarrow v$
12: $\quad\quad\quad$ **if** $v = (r \bmod 2)$ **then** decide(v)
13: $\quad\quad$ **else** $est \leftarrow (r \bmod 2)$
14: $\quad\quad r \leftarrow r + 1$

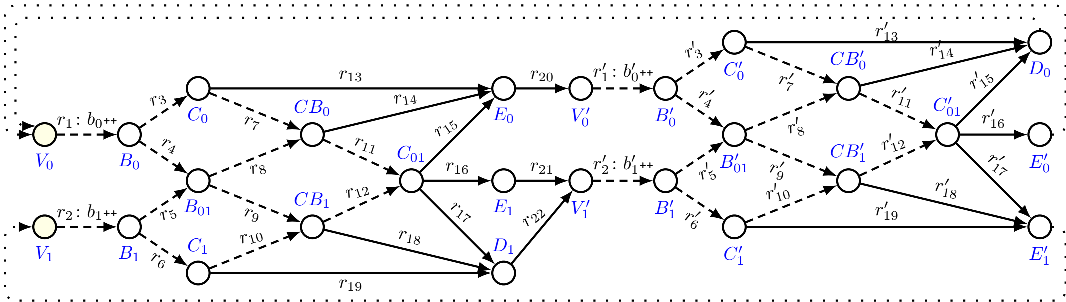


Figure 3 The naive threshold automaton of the Byzantine consensus of Algorithm 1 where the embedded bv-broadcast automaton is depicted with dashed arrows. Precise formulations of all rules are in Appendix D. Note that the rules r_{20}, r_{21} and r_{22} represent transitions from the end of an odd round to the beginning of the following (even) round of Algorithm 1, while the dotted edges represent transitions from the end of an even round to the beginning of the following (odd) one.

As mentioned in Section 2, recall that the algorithm is communication-closed, so that for simplicity in the presentation we omit the current round number r as the subscript of the variables and the parameter of the function calls. Variable *favorites* is an array of n indices whose j^{th} slot records, upon delivery, the message broadcast by process j in the current round. Each process p_i manages the following local variables: the current estimate *est*, initially the input value of p_i; and a set of binary values *qualifiers*. This algorithm maintains a round number r, initially 0 (line 4), and incremented at the end of each iteration of the loop at line 14. Process p_i exchanges estimate (EST) and auxiliary (AUX) messages (lines 6–8), until it receives AUX messages from $n - t$ distinct processes whose values were bv-delivered by p_i (line 9). Process p_i then tries at line 12 to decide a value v that depends on the content of *qualifiers* and the parity of the round. If *qualifiers* is a singleton there are two possible cases: if the value is the parity of the round then p_i decides this value (line 12), otherwise it sets its estimate to this value (line 11). If *favorites* contains both binary values, then p_i sets its estimate to the parity of the round (line 13). Although p_i does not exit the infinite loop to help other processes decide, it can safely exit the loop after two rounds at the end of the second round that follows the first decision because all processes will be guaranteed to have decided. Note that even though a process may invoke decide($\cdot$) multiple times at line 12, only the first decision matters as the decided value does not change (see Section 5).

The effect of fairness. Note that the fairness notion from Section 3.3 ensures there is a round r in which all correct processes bv-deliver (r mod 2) first. The following lemma states that under the fairness assumption there is a round of Algorithm 1 in which all correct processes start with the same estimate. The proof is deferred to Appendix C.

▶ **Lemma 4.** *If the infinite sequence of bv-broadcast executions of Algorithm 1 is fair, with the rth execution being (r mod 2)-good, then all correct processes start round $r+1$ of Algorithm 1 with estimate r mod 2.*

Modeling deterministic consensus. Figure 3 depicts the threshold automaton (TA) obtained by modeling Algorithm 1 with the method we detailed in Section 3.1. The TA depicts two iterations of the repeat loop (line 5), since Algorithm 1 favors different values depending on the parity of the round number. For simplicity, we refer to the concatenation of two consecutive rounds of the algorithm as a *superround* of the TA. As one can expect, this TA embeds the TA of the bv-broadcast which is depicted by the dashed arrows, just as

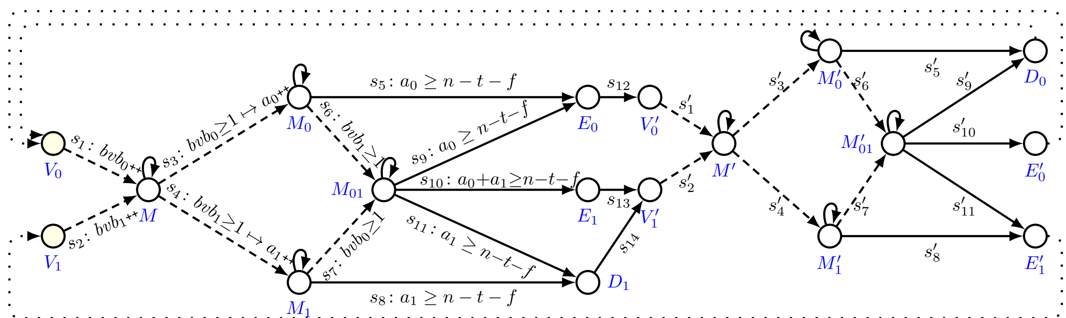


Figure 4 The simplified threshold automaton of the Byzantine consensus of Algorithm 1 obtained after model checking the bv-broadcast. Rules s'_j, $1 \leq i \leq 11$, are obtained from s_j by replacing each variable $c \in \{a_0, a_1, bvb_0, bvb_1\}$ with its corresponding one c'.

Algorithm 1 invokes the bv-broadcast algorithm of Fig. 1. We thus distinguish the outer TA modeling the consensus algorithm from the inner TA modeling the bv-broadcast algorithm. Although Algorithm 1 is relatively simple, the global TA happens to be too large to be verified through model checking, as we explain in Section 6; the main limiting factor is its 14 unique guards that constrain the variables to enable rules in the TA. The detail of each rule of the TA is deferred to Appendix D.

4.2 Simplified threshold automaton

Our objective is to formally prove that Algorithm 1 is unconditionally safe, and that it is live under the assumption of fairness at the bv-broadcast level. Since the threshold automaton of Figure 3 is too large to be handled automatically, we build on the properties proved for the bv-broadcast to simplify in the threshold automaton from Figure 3 the part representing the bv-broadcast. On the resulting simpler threshold automaton, assuming fairness of the bv-broadcast, we prove the termination of Algorithm 1 with ByMC in Section 6.

High-level idea. Ideally, the simplified threshold automaton could be obtained from the one of Fig. 3 by merging all internal states of the bv-broadcast into a single state with two possible outcomes. However, such a merge is not trivial because the bv-broadcast procedure "leaks" into the consensus algorithm. First of all, line 7 of Algorithm 1 refers to contestants, a global variable that is modified by the bv-broadcast algorithm (Fig. 1). Second, a process can execute line 8 of Algorithm 1 even if the bv-broadcast has not terminated. To capture this porosity, we introduce a new shared variable, some additional states and a transition rule that exploits a correctness property of the bv-broadcast.

A superround R of the simplified automaton from Fig. 4 captures round $2R{-}1$ followed by round $2R$ of Algorithm 1. One can thus restate Lemma 4 as the following corollary in the TA terminology. The proof is deferred to Appendix E.

▶ **Corollary 5.** *Let $r \in \mathbb{N}$ be such that the rth execution of* bv-broadcast *in Algorithm 1 is* $(r \bmod 2)$*-good. Then:*

- *If there exists $R \in \mathbb{N}$ with $r = 2R{-}1$, then $\Box\left(\kappa[M_0, R] = 0\right)$ holds.*
- *If there exists $R \in \mathbb{N}$ with $r = 2R$, then $\Box\left(\kappa[M'_1, R] = 0\right)$ holds.*

5 Verification of Byzantine Consensus

In this section we formally prove that Algorithm 1 solves the Byzantine consensus problem with the fair bv-broadcast and without partial synchrony. (Appendix B provides a counter-example illustrating why the algorithm does not terminate without the fair broadcast.) In particular, we apply a methodology developed for crash fault tolerant randomized consensus [10] to our context to prove both the safety (Section 5.1) and liveness (Section 5.2) properties of the deterministic Byzantine consensus algorithm.

5.1 Safety

Under no fairness assumption, one can prove the safety properties – agreement and validity – of the Byzantine consensus based on bv-broadcast. Precisely, we formulate these properties in LTL and want to establish that they hold on the threshold automaton of Fig. 4.

Agreement requires that no two correct processes disagree, that is, if one process decides v then no process should decide $1-v$ for all binary values $v \in \{0,1\}$. Thus, we want to prove that the following formula holds for both values of v:

$$\forall R \in \mathbb{N}, \forall R' \in \mathbb{N} \left(\Diamond \, \kappa[D_v, R] \neq 0 \;\Rightarrow\; \Box \, \kappa[D_{1-v}, R'] = 0 \right) \;, \qquad (\mathit{Agree}_v)$$

stating that for any two superrounds R and R', if eventually a process decides v, then globally (in any superround) no process will decide $1-v$. In terms of the TA from Fig. 4, if a process enters location D_v no process should enter location D_{1-v} (in that superround or any other).

Validity requires that if no process proposes a value $v \in \{0,1\}$, no process should ever decide that value. Hence, we want to prove the following formula for both values of v:

$$\forall R \in \mathbb{N} \left(\kappa[V_v, 1] = 0 \;\Rightarrow\; \Box \, \kappa[D_v, R] = 0 \right) \;, \qquad (\mathit{Valid}_v)$$

stating that if initially no process has value v, then globally (in any superround) no process decides v. In terms of the TA, if location V_v is initially empty (in superround 1), then no process should enter location D_v in any superround.

ByMC can only check formulas of the form $\forall R \in \mathbb{N} \; \varphi[R]$ (see Appendix A). Thus, automatically checking (Agree_v) and (Valid_v) is non-trivial, as they both involve two superround numbers: R and R' in (Agree_v), and 1 and R in (Valid_v). We instead check well-chosen one-superround invariants $(\mathit{Inv1}_v)$ and $(\mathit{Inv2}_v)$:

$$\forall R \in \mathbb{N} \left(\Diamond \, \kappa[D_v, R] \neq 0 \;\Rightarrow\; \Box \left(\kappa[D_{1-v}, R] = 0 \wedge \kappa[E'_{1-v}, R] = 0 \right) \right) \;, \qquad (\mathit{Inv1}_v)$$

$$\forall R \in \mathbb{N} \left(\Box \, \kappa[V_v, R] = 0 \;\Rightarrow\; \Box \left(\kappa[D_v, R] = 0 \wedge \kappa[E'_v, R] = 0 \right) \right) \;. \qquad (\mathit{Inv2}_v)$$

The choice of these invariants follows a previous approach used for the crash fault tolerant consensus where it is shown that these invariants imply (Agree_v) and (Valid_v) [10, Proposition 2]. Intuitively, this follows from the fact that (i) emptiness of D_0 and E'_0 in one superround leads to the emptiness of V_0 in the next superround, and (ii) emptiness of E'_1 (and D_1) in one superround leads to the emptiness of V_1 in the next superround. Therefore, in order to prove agreement and validity, we only need to prove $(\mathit{Inv1}_v)$ and $(\mathit{Inv2}_v)$ for both values $v \in \{0,1\}$. We successfully do this automatically with ByMC (see Section 6).

5.2 Liveness

We now aim at proving termination of Algorithm 1. First, we need to prove that every superround eventually terminates, in the sense that for every round eventually there are no processes in any location of that round with the exception of the final ones (D_0, E'_0 and E'_1). Formally, using ByMC we prove the following:

$$\forall R \in \mathbb{N} \lozenge \Big(\bigwedge_{L \in \mathcal{L} \setminus \{D_0, E'_0, E'_1\}} \kappa[L, R] = 0 \Big) \ . \qquad (SRoundTerm)$$

From this property and the shape of the TA from Fig. 4, it easily follows that if no process ever enters E'_0 and E'_1 of some superround, then all processes visit D_0 in that superround. Similarly, if no process ever enters E_0 and E_1 of some superround, then all processes visit D_1 in that superround. This allows us to express termination as the following LTL property on the threshold automaton of Fig. 4:

$$\exists R \in \mathbb{N} \Big(\Box \big(\kappa[E_0, R] = 0 \land \kappa[E_1, R] = 0 \big) \ \lor \ \Box \big(\kappa[E'_0, R] = 0 \land \kappa[E'_1, R] = 0 \big) \Big) \ . \quad (Term)$$

In words, there is a superround R in which either (i) all processes visit D_1, or (ii) all processes visit D_0. Here again formula ($Term$) is non-trivial to check since it contains an existential quantifier over superrounds, that cannot be handled by the model checker ByMC. Adapting the technique from [10, Section 7] to a non-randomized context, it is sufficient to prove a couple of properties on the threshold automaton of Fig. 4, that we detail below. The first property expresses that if no process starts a superround R with value v, then all processes decide $1-v$ in superround R:

$$\begin{aligned} \forall R \in \mathbb{N} & \Big(\Box \big(\kappa[V_0, R] = 0 \big) \Rightarrow \Box \big(\kappa[E_0, R] = 0 \land \kappa[E_1, R] = 0 \big) \Big) \\ \land & \ \Big(\Box \big(\kappa[V_1, R] = 0 \big) \Rightarrow \Box \big(\kappa[E'_0, R] = 0 \land \kappa[E'_1, R] = 0 \big) \Big) \ . \end{aligned} \qquad (Dec)$$

The second property claims that (i) emptiness of M_0 in superround R implies (emptiness of E_0 and therefore also) emptiness of D_0 and E'_0 in R and (ii) emptiness of M'_1 in superround R implies emptiness of E'_1 in R:

$$\begin{aligned} \forall R \in \mathbb{N} \Big(& (\Box \, \kappa[M_0, R] = 0) \ \Rightarrow \ \Box \, (\kappa[D_0, R] = 0 \land \kappa[E'_0, R] = 0)) \\ & \land \ (\Box \, \kappa[M'_1, R] = 0) \ \Rightarrow \ \Box \, \kappa[E'_1, R] = 0) \Big) \ . \end{aligned} \qquad (Good)$$

The main idea is to exploit the fairness of bv-broadcast, which ensures the existence of a round r which is (r mod 2)-good. Intuitively, the next superround $R = \lceil r/2 \rceil$ is the desired witness for ($Term$), namely the one in which all processes decide (not necessarily for the first time). We formalize this in our main result:

▶ **Theorem 6.** *Assuming fairness of the* bv-broadcast*, Algorithm 1 terminates.*

Proof. First we prove formulas ($SRoundTerm$) and (Dec) and ($Good$) automatically using the model checker ByMC. Formula ($SRoundTerm$) guarantees that formula ($Term$) indeed expresses termination. Next, we show that formulas (Dec) and ($Good$) together imply ($Term$). Indeed, since we assume fairness of the bv-broadcast, from Corollary 5 we know that there is a superround R in which one of the following two scenarios happen:

- $\Box\,\kappa[M_1', R] = 0$. In this case formula (*Good*) implies $\Box\,\kappa[E_1', R] = 0$. Note that the form of the (dotted) round-switch rules yield that no process starts the superround $R{+}1$ with value 1, that is, we have $\Box\,\kappa[V_1, R{+}1] = 0$. Then formula (*Dec*) implies $\Box\,\big(\kappa[E_0', R{+}1] = 0 \wedge \kappa[E_1', R{+}1] = 0\big)$, which makes formula (*Term*) true, that is, all processes visit D_0 in superround $R{+}1$.
- $\Box\,\kappa[M_0, R] = 0$. In this case formula (*Good*) implies $\Box\,\big(\kappa[D_0, R] \wedge \kappa[E_0', R] = 0)\big)$. Now the round-switch rules yield that no process starts the superround $R{+}1$ with value 0, that is, we have $\Box\,\kappa[V_0, R{+}1] = 0$. Then formula (*Dec*) implies $\Box\,\big(\kappa[E_0, R{+}1] = 0 \wedge \kappa[E_1, R{+}1] = 0\big)$, which satisfies formula (*Term*), that is, all processes visit D_1 in $R{+}1$.

As a consequence, our automated proofs of properties (*SRoundTerm*) and (*Dec*) and (*Good*) guarantee termination of Algorithm 1 under fairness of bv-broadcast. ◀

6 Experiments

In this section, we model check the safety but also the liveness properties of Byzantine consensus for any parameters t and $n > 3t$. In particular, we show that we formally verify the simplified representation of the blockchain consensus in less than 70 seconds, whereas we could not model check its naive representation.

Experimental settings. We used the parallelized version of ByMC 2.4.4 with MPI. The bv-broadcast and the simplified automaton were verified on a laptop with Intel® Core™ i7-1065G7 CPU @ 1.30GHz × 8 and 32 GB of memory. The naive Threshold Automaton (TA) timed-out even on a 4 AMD Opteron 6276 16-core CPU with 64 cores at 2300MHz with 64 GB of memory. *Good* and *Dec* are only relevant for the simplified automaton. The specification of the termination for ByMC is deferred to Appendix F.

Table 2 Although none of the properties of the naive blockchain consensus could be verified within a day of execution of the model checker, it takes about ∼4 s to verify each property on the simplified representation of the blockchain consensus. Overall it takes less than 70 seconds to verify both that the binary value broadcast and the simplified representation of the blockchain consensus are correct.

TA	Size	Property	# schemas	Avg. length	Time
bv-broadcast (Fig. 2)	4 unique guards, 10 locations, 19 rules	*BV-Just$_0$*	90	54	5.61s
		BV-Obl$_0$	90	79	6.87s
		BV-Unif$_0$	760	97	27.64s
		BV-Term	90	79	6.75s
Naive consensus (Fig. 3)	14 unique guards, 24 locations, 45 rules	*Inv1$_0$*	>100 000	-	>24h
		Inv2$_0$	>100 000	-	>24h
		SRound-Term	>100 000	-	>24h
Simplified consensus (Fig. 4)	10 unique guards, 16 locations, 37 rules	*Inv1$_0$*	6	102	4.68s
		Inv2$_0$	2	73	4.56s
		SRound-Term	2	109	4.13s
		Good$_0$	2	67	4.55s
		Dec$_0$	2	73	4.62s

Results. Table 2 depicts the time (6th column) it takes to verify each property (3rd column) automatically. In particular, it lists the TA (1st column) on which these properties were tested, as well as the size of these TA (2nd column) as the number of guards locations and rules they contain. A schema (4th column) is a sequence of unlocked guards (contexts) and rule sequences that is used to generate execution paths [37] whose average length appears in the 5th column. It demonstrates the efficiency of our approach as it allows to verify all properties of the Byzantine consensus automatically in less than 70 seconds whereas a non-compositional approach timed out. Although not indicated here, we also generated a counter-example of $Inv1_0$ for $n > 3t$ on the composite automaton in ∼4 s.

7 Related Work

Interactive theorem provers [62, 59, 70] already checked proofs of algorithmic components used in the blockchain industry. In particular, Coq helped prove the Raft consensus algorithm [71] and parts of crash fault tolerant distributed ledgers [12, 5], neither of which is Byzantine fault tolerant, but also some safety properties of PBFT [60] and of the Byzantine consensus algorithm of the Algorand blockchain [3]. In addition, Dafny [31] proved MultiPaxos, a consensus algorithm that tolerates crash failures. Isabelle/HOL [54] was used to prove Byzantine fault tolerant algorithms [17] and was combined with Ivy to prove a simplified variant of the Byzantine consensus [45] of Stellar [44] but without its dynamic quorum system [46]. Theorem provers check proofs, not the algorithms. Hence, one has to invest efforts into writing detailed mechanical proofs.

Specialized decision procedures are a way of proving consensus algorithms. They were used to prove Paxos [40], which could itself be used in the aforementioned crash fault tolerant distributed ledgers. Crash fault tolerant consensus algorithms were manually encoded with their invariants and properties to prove formulae using the Z3 SMT solver [24]. Decision procedures also proved the safety of Byzantine fault tolerant consensus algorithms when $f = t$ [9] but not their termination. Similarly, a proof by refinement of the safety of a Byzantine variant of Paxos was proposed [41] but its liveness is not proven. These decision procedures require the user to fit the specification into a suitable logical fragment.

Explicit-state model checking fully automates verification of distributed algorithms [32, 72]. It allows to check the reliable broadcast algorithm [33], a common component of various blockchain consensus algorithms [47, 19, 18]. TLC [72] checked a reduction of fault tolerant distributed algorithms in the Heard-Of model that exploits their communication-closed property [16]. And the agreement of consensus algorithms was proved in the asynchronous setting [55]. These tools enumerate all reachable states and suffer from state explosion.

Symbolic model checkers [13] cope with this explosion by representing state transitions efficiently. NuSMV and SAT helped check consensus algorithms for up to 10 processes [68, 69]. Apalache [34] uses satisfiability modulo theories (SMT) to check inductive invariants and verify symbolic executions of TLA$^+$ specifications of the reliable broadcast and crash fault tolerant consensus algorithms but requires parameters to be fixed. These tools cannot be used to prove (or disprove) correctness for an arbitrary number of processes.

Parameterized model checking [23] works for an arbitrary number n of processes [11]. Although the problem is undecidable [6] in general, one can verify specific classes of algorithms [28]. Indeed, distributed algorithms with a ring-based topology were checked with automata-theoretic method [2] and with Presburger arithmetics formulae verified by an SMT solver [61]. Bosco [63] has been the focus of various parameterized verification techniques [42, 7], however, it acts as a fast path wrapper around a separate correct consensus

algorithm that remains itself to be proven. The condition-based consensus algorithm [53, 52] was verified [7] with the Byzantine model checker ByMC [37, 39, 35], only under the condition that the difference between the numbers of processes initialized with 0 and 1 differ by at least t. Recently, almost-sure termination was proved by assuming a round-rigid adversary [10], but this is insufficient to prove our termination. In this paper, we also exploit ByMC but prove the Byzantine consensus algorithm [19] of an existing blockchain [20].

8 Conclusion

We presented the first formal verification of a blockchain consensus algorithm thanks to a new holistic approach. By modeling directly the pseudocode into a disambiguated threshold automaton we guarantee that the "actual" algorithm is verified. By model checking the threshold automaton for any parameters without the need for user-defined invariants and proofs, we drastically reduce the risks of human errors.

References

1. Ittai Abraham, Guy Golan Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance. Technical report, arXiv, December 2017. `arXiv:1712.01367`.
2. C. Aiswarya, Benedikt Bollig, and Paul Gastin. An automata-theoretic approach to the verification of distributed algorithms. *Information and Computation*, 259(3):305–327, 2018.
3. Musab A. Alturki, Jing Chen, Victor Luchangco, Brandon M. Moore, Karl Palmskog, Lucas Peña, and Grigore Rosu. Towards a verified model of the algorand consensus protocol in Coq. In *International Workshops on Formal Methods (FM'19)*, pages 362–367, 2019.
4. Rajeev Alur and Thomas A. Henzinger. Finitary fairness. In *Annual IEEE Symposium on Logic in Computer Science (LICS'94)*, pages 52–61. IEEE Computer Society Press, 1994.
5. Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In *ACM European Conference on Computer Systems (CCS'18)*, 2018.
6. Krzysztof R Apt and Dexter C Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, May 1986.
7. A. R. Balasubramanian, Javier Esparza, and Marijana Lazic. Complexity of verification and synthesis of threshold automata. In *International Symposium on Automated Technology for Verification and Analysis (ATVA'20)*, pages 144–160, 2020.
8. Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Annual ACM Symposium on Principles of Distributed Computing (PODC'83)*, pages 27–30, 1983.
9. Idan Berkovits, Marijana Lazic, Giuliano Losa, Oded Padon, and Sharon Shoham. Verification of threshold-based distributed algorithms by decomposition to decidable logics. In *International Conference on Computer Aided Verification (CAV'19)*, pages 245–266, 2019.
10. Nathalie Bertrand, Igor Konnov, Marijana Lazic, and Josef Widder. Verification of randomized consensus algorithms under round-rigid adversaries. In *International Conference on Concurrency Theory (CONCUR'19)*, pages 33:1–33:15, 2019.
11. Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. *Decidability of Parameterized Verification*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.
12. Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: An introduction. *R3 CEV, August*, 2016.

13 Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10ˆ20 states and beyond. In *Annual Symposium on Logic in Computer Science (LICS'90)*, pages 428–439, 1990.

14 Christian Cachin, Daniel Collins, Tyler Crain, and Vincent Gramoli. Anonymity preserving Byzantine vector consensus. In *European Symposium on Research in Computer Security (ESORICS'20)*, pages 133–152, September 2020.

15 Christian Cachin and Luca Zanolini. Asymmetric Byzantine consensus. Technical Report 2005.08795, arXiv, 2020.

16 Mouna Chaouch-Saad, Bernadette Charron-Bost, and Stephan Merz. A reduction theorem for the verification of round-based distributed algorithms. In *International Workshop on Reachability Problems (RP'09)*, pages 93–106, 2009.

17 Bernadette Charron-Bost, Henri Debrat, and Stephan Merz. Formal verification of consensus algorithms tolerating malicious faults. In *SSS*, pages 120–134, 2011.

18 Pierre Civit, Seth Gilbert, and Vincent Gramoli. Polygraph: Accountable Byzantine agreement. In *Proceedings of the 41st IEEE International Conference on Distributed Computing Systems (ICDCS'21)*, July 2021.

19 Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: Efficient leaderless Byzantine consensus and its applications to blockchains. In *International Symposium on Network Computing and Applications (NCA'18)*, 2018. URL: http://gramoli.redbellyblockchain.io/web/doc/pubs/DBFT-preprint.pdf.

20 Tyler Crain, Christopher Natoli, and Vincent Gramoli. Red Belly: A secure, fair and scalable open blockchain. In *IEEE Symposium on Security and Privacy (S&P'21)*, 2021.

21 Andrei Damian, Cezara Drăgoi, Alexandru Militaru, and Josef Widder. Communication-closed asynchronous protocols. In *International Conference on Computer Aided Verification (CAV'19)*, pages 344–363, 2019.

22 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'08)*, pages 337–340, 2008.

23 Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999.

24 Cezara Dragoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. A logic-based framework for verifying consensus algorithms. In *International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'14)*, pages 161–181, 2014.

25 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

26 Ali Ebnenasir and Alex P. Klinkhamer. Topology-specific synthesis of self-stabilizing parameterized systems with constant-space processes. *IEEE Trans. Software Eng.*, 47(3):614–629, 2021.

27 Tzilla Elrad and Nissim Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming*, 2(3):155–173, 1982.

28 E. Allen Emerson and Vineet Kahlon. Reducing model checking of the many to the few. In *International Conference on Automated Deduction (CADE'17)*, pages 236–254, 2000.

29 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

30 Dana Fisman, Orna Kupferman, and Yoad Lustig. On verifying fault tolerance of distributed protocols. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'08)*, pages 315–331, 2008.

31 Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *Symposium on Operating Systems Principles (SOSP'15)*, pages 1–17, 2015.

32 Gerard Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

33 Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Towards modeling and model checking fault-tolerant distributed algorithms. In *International Symposium on Model Checking Software (SPIN'13)*, volume 7976 of *LNCS*, pages 209–226, 2013.

34 Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. TLA+ model checking made symbolic. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):123:1–123:30, 2019.

35 Igor Konnov, Marijana Lazic, Ilina Stoilkovska, and Josef Widder. Tutorial: Parameterized verification with byzantine model checker. In *International Conference on Formal Techniques for (Networked and) Distributed Systems (FORTE'20)*, pages 189–207, 2020.

36 Igor Konnov, Marijana Lazic, Helmut Veith, and Josef Widder. Para^2: parameterized path reduction, acceleration, and SMT for reachability in threshold-guarded distributed algorithms. *Formal Methods in System Design*, 51(2):270–307, 2017.

37 Igor Konnov, Marijana Lazic, Helmut Veith, and Josef Widder. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In *Symposium on Principles of Programming Languages (POPL'17)*, pages 719–734. ACM, 2017.

38 Igor Konnov, Helmut Veith, and Josef Widder. On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. *Information and Computation*, 252:95–109, 2017.

39 Igor Konnov and Josef Widder. ByMC: Byzantine model checker. In *ISoLA*, pages 327–342, 2018.

40 Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. Inductive sequentialization of asynchronous programs. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'20)*, pages 227–242, 2020.

41 Leslie Lamport. Byzantizing paxos by refinement. In *International Symposium on Distributed Computing (DISC'11)*, pages 211–224, 2011.

42 Marijana Lazic, Igor Konnov, Josef Widder, and Roderick Bloem. Synthesis of distributed algorithms with parameterized threshold guards. In *International Conference on Principles of Distributed Systems (OPODIS'17)*, pages 32:1–32:20, 2017.

43 Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

44 Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 80–96, 2019.

45 Giuliano Losa and Mike Dodds. On the formal verification of the stellar consensus protocol. In *Workshop on Formal Methods for Blockchains (FMBC@CAV'20)*, pages 9:1–9:9, 2020.

46 Giuliano Losa, Eli Gafni, and David Mazières. Stellar consensus by instantiation. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing, DISC 2019*, volume 146 of *LIPIcs*, pages 27:1–27:15, 2019.

47 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *Conference on Computer and Communications Security (CCS'16)*, 2016.

48 Nahal Mirzaie, Fathiyeh Faghih, Swen Jacobs, and Borzoo Bonakdarpour. Parameterized synthesis of self-stabilizing protocols in symmetric networks. *Acta Informatica*, 57(1-2):271–304, 2020.

49 Hadi Moloodi, Fathiyeh Faghih, and Borzoo Bonakdarpour. Parameterized distributed synthesis of fault-tolerance using counter abstraction. In *Proceedings of the 40th International Symposium on Reliable Distributed Systems (SRDS)*, pages 67–77. IEEE, 2021.

50 Achour Mostéfaoui, Hamouna Moumen, and Michel Raynal. Signature-free asynchronous Byzantine consensus with $T < N/3$ and $O(N^2)$ messages. In *Symposium on Principles of Distributed Computing (PODC'14)*, pages 2–9, 2014.

51 Achour Mostéfaoui, Hamouna Moumen, and Michel Raynal. Signature-free asynchronous binary Byzantine consensus with $t < n/3, O(n^2)$ messages and $O(1)$ expected time. *Journal of the ACM*, 2015.

52 Achour Mostéfaoui, Eric Mourgaya, Philippe Raipin Parvédy, and Michel Raynal. Evaluating the condition-based approach to solve consensus. In *Dependable Systems and Networks (DSN'03)*, pages 541–550, 2003.

53 Achour Mostéfaoui, Sergio Rajsbaum, and Michel Raynal. Conditions on input vectors for consensus solvability in asynchronous distributed systems. *Journal of the ACM*, 50(6):922–954, November 2003.

54 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

55 Tatsuya Noguchi, Tatsuhiro Tsuchiya, and Tohru Kikuno. Safety verification of asynchronous consensus algorithms with model checking. In *International Symposium on Dependable Computing (PRDC'12)*, pages 80–88, 2012.

56 Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

57 Amir Pnueli. The temporal logic of programs. In *IEEE Annual Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–57, 1977.

58 Mojżesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du congrès de Mathématiciens des Pays Slaves*, pages 92–101, 1929.

59 Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. Formal specification, verification, and implementation of fault-tolerant systems using EventML. *Electronic Communication of the European Association of Software Science and Technology*, 72, 2015.

60 Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Jorge Esteves Veríssimo. Velisarios: Byzantine fault-tolerant protocols powered by coq. In Amal Ahmed, editor, *Proceedings of the 27th European Symposium on Programming (ESOP)*, volume 10801 of *Lecture Notes in Computer Science*, pages 619–650. Springer, 2018.

61 Arnaud Sangnier, Nathalie Sznajder, Maria Potop-Butucaru, and Sébastien Tixeuil. Parameterized verification of algorithms for oblivious robots on a ring. *Formal Methods in System Design*, 56(1):55–89, 2020.

62 Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Proceedings of the ACM on Programming Languages*, 2(POPL):28:1–28:30, 2018.

63 Yee Jiun Song and Robbert van Renesse. Bosco: One-step Byzantine asynchronous consensus. In *International Symposium on Distributed Computing (DISC'08)*, pages 438–450, 2008.

64 Ilina Stoilkovska, Igor Konnov, Josef Widder, and Florian Zuleger. Eliminating message counters in threshold automata. In *International Symposium on Automated Technology for Verification and Analysis (ATVA'20)*, pages 196–212, 2020.

65 Pierre Sutra. On the correctness of egalitarian paxos. *Information Processing Letters*, 156:105901, 2020. doi:10.1016/j.ipl.2019.105901.

66 Amer Tahat and Ali Ebnenasir. A hybrid method for the verification and synthesis of parameterized self-stabilizing protocols. In Maurizio Proietti and Hirohisa Seki, editors, *24th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR)*, volume 8981 of *Lecture Notes in Computer Science*, pages 201–218. Springer, 2014.

67 Pierre Tholoniat and Vincent Gramoli. Formal verification of blockchain Byzantine fault tolerance. In *Workshop on Formal Reasoning in Distributed Algorithms (FRIDA'19)*, October 2019. arXiv:1909.07453.

68 Tatsuhiro Tsuchiya and André Schiper. Using bounded model checking to verify consensus algorithms. In Gadi Taubenfeld, editor, *Distributed Computing*, pages 466–480, 2008.

69 Tatsuhiro Tsuchiya and André Schiper. Verification of consensus algorithms using satisfiability solving. *Distributed Computing*, 23(5-6):341–358, 2011.

70 Klaus von Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend synchrony: synchronous verification of asynchronous distributed programs. *Proceedings of the ACM on Programming Languages*, 3(POPL):59:1–59:30, 2019.

71 James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'15)*, pages 357–368, 2015.

72 Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA$^+$ specifications. In *CHARME*, pages 54–66, 1999.

A Reducing multi-round TA to one-round TA

Let us first formally define a (finite or infinite) *run* in a (one-round or multi-round) counter system *Sys*(TA). It is an alternating sequence of configurations and transitions $\sigma_0, t_1, \sigma_1, t_2, \dots$ such that $\sigma_0 \in I$ is an initial configuration and for every $i \geq 1$ we have that t_i is unlocked in σ_{i-1}, and executing it leads to σ_i, denoted $t_i(\sigma_{i-1}) = \sigma_i$.

Here we briefly describe the reasoning behind the reduction of multi-round TAs to one-round TAs [10, Theorem 6]. Note that the behavior of a process in one round only depends on the variables (the number of messages) of that round. Namely, we check if a transition is unlocked in a round by evaluating a guard and a location counter in that round. This allows us to modify a run by swapping two transitions from different rounds, as they do not affect each other, and preserve LTL$_{\text{-X}}$ properties, which are properties expressed in LTL without the next operator **X**. The type of swapping we are interested in is the one where a transition of round R is followed by a transition of round $R' < R$. Starting from any (fully asynchronous) run, if we keep swapping all such pairs of transitions, we will obtain a run in which processes synchronize at the end of each round and which has the same LTL$_{\text{-X}}$ properties as the initial one. This, so-called *round-rigid* structure, allows us to isolate a single round and analyze it. Still, different rounds might behave differently as they have different initial configurations. If we have a formula $\forall R \in \mathbb{N}.\ \varphi[R]$, where $\varphi[R]$ is in the above mentioned fragment of (multi-round) LTL, then Theorem 6 of [10] shows exactly that it is equivalent to check that (i) this formula holds (or $\varphi[R]$ holds on all rounds R) on a multi-round TA, and (ii) formula $\varphi[1]$ (or just φ) holds on the one-round TA$'$ (naturally obtained from the TA by removing dotted round-switch rules) with respect to all possible initial configurations of all rounds. Thus, we can verify properties of the form $\forall R \in \mathbb{N}.\ \varphi[R]$ on multi-round threshold automata, by using ByMC to check φ on a one-round threshold automaton with an enlarged set of initial configurations.

B Examples of fairness and of non-termination without fairness

First, we explain that the fairness is satisfied as soon as one execution of bv-broadcast has correct processes delivering all values broadcast by correct processes first. Then, we explain that the Byzantine consensus algorithm cannot terminate without an additional assumption, like fairness.

Relevance of the fairness assumption

It is interesting to note that our fairness assumption is satisfied by the existence of an execution with a particular reception order of some messages of the two broadcasts within the bv-broadcast. Consider that $t = \lceil n/3 \rceil - 1$ and that at the beginning of a round r, the two following properties hold: (i) estimate $r \bmod 2$ is more represented than estimate $(1-r) \bmod 2$ among correct processes and (ii) all correct processes deliver the values broadcast by correct processes before any value broadcast during the bv-broadcast by Byzantine processes are

delivered. Indeed, the existence of such a round r in any infinite sequence of executions of bv-broadcast implies that this sequence is fair (Def. 3): as $r \bmod 2$ is the only value that can be broadcast by $t{+}1$ correct processes, this is the first value that is received from $t{+}1$ distinct processes and rebroadcast by the rest of the correct processes. This is thus also the first value that is bv-delivered by all correct processes.

Non-termination without fairness

It is interesting to note why Algorithm 1 does not solve consensus when $t < n/3$ and without our fairness assumption. We exhibit an example of execution of the algorithm with $n = 4$ and $f = 1$, starting at round r and for which the estimates of the correct processes are kept as $(1-r) \bmod 2, (1-r) \bmod 2, r \bmod 2$ in rounds r and $r{+}2$. Repeating this while incrementing r yields an infinite execution, so that the algorithm never terminates.

▶ **Lemma 7.** *Algorithm 1 does not terminate without fairness.*

Proof. Consider, for example, processes p_1, p_2, p_3 and p_4 where p_4 is Byzantine and where $0, 0, 1$ are the input values of the correct processes p_1, p_2, p_3, respectively, at round 1. We show that at the beginning of round 2, p_1, p_2, p_3 have estimates $0, 1, 1$. First, as a result of the broadcast (line 2), consider that p_1 and p_2 receive 0 from p_1, p_2 and p_4 so that p_1, p_2 bv-deliver 0. Second, p_2 and p_3 receive 1 from p_3, p_4 and finally p_2 so that p_2, p_3 bv-deliver 1. Third, p_3 receives 0 from p_0, p_2 and finally from p_3 itself, hence p_3 bv-delivers 0. Now we have: (a) p_1, p_2, p_3 bv-deliver 0, 0, 1 and (b) p_2, p_3 later bv-deliver 1 and 0, respectively. As a result of (a), we have p_1, p_2 broadcast, and say p_4 sends, $\langle \text{AUX}, 0, \cdot\rangle$ so that p_0 receives these three messages, p_1, p_2 broadcast $\langle \text{AUX}, 0, \cdot\rangle$, and say p_4 sends, $\langle \text{AUX}, 1, \cdot\rangle$ to p_2 so that p_2 receives these messages, p_1 broadcasts $\langle \text{AUX}, 0, \cdot\rangle$ while p_3 broadcasts, and say p_4 sends, $\langle \text{AUX}, 1, \cdot\rangle$ so that p_3 receives these messages. Finally, by (b) we have $\mathit{contestants}_2 = \mathit{contestants}_3 = \{0, 1\}$. This implies that the $n{-}t$ first values inserted in $\mathit{favorites}_1$, $\mathit{favorites}_2$ and $\mathit{favorites}_3$ in round r are values $\{0\}$, $\{0, 1\}$, $\{0, 1\}$, respectively. Finally, $\mathit{qualifiers}_1$, $\mathit{qualifiers}_2$ and $\mathit{qualifiers}_3$ are $\{0\}$, $\{0, 1\}$ and $\{0, 1\}$, respectively. And p_1, p_2, p_3 set their estimate to $0, 1, 1$.

It is easy to see that a symmetric execution in round $r' = r + 1$ leads processes to change their estimate from $0, 1, 1$ to $0, 0, 1$ looping back to the state where $r \bmod 2 = 1$ and estimate are $(1-r) \bmod 2, (1-r) \bmod 2, r \bmod 2$. ◀

C Starting a round with identical estimate

▶ **Lemma 8** (Lemma 4). *If the infinite sequence of bv-broadcast invocations of Algorithm 1 is fair, with the rth invocation (in round r) being $(r \bmod 2)$-good, then all correct processes start round $r{+}1$ of Algorithm 1 with estimate $r \bmod 2$.*

Proof. The argument is that all correct processes wait until a growing prefix of the bv-delivered values that are re-broadcast implies that there is a subset of favorites, called *qualifiers*, containing messages from $n - t$ distinct processes such that $\forall v \in \mathit{qualifiers}.\ v \in \mathit{contestants}$. As we assume that the infinite sequence of bv-broadcast invocations of Algorithm 1 is fair, with the rth invocation being $(r \bmod 2)$-good, then we know that in round r for every pair of correct processes p_i and p_j we have $p_i.\mathit{qualifiers} \subseteq p_j.\mathit{qualifiers}$ or $p_j.\mathit{qualifiers} \subseteq p_i.\mathit{qualifiers}$. If $p_i.\mathit{qualifiers} = p_j.\mathit{qualifiers}$ for all pairs, then by examination of the code, we know that they will set their estimate *est* to the same value depending on the parity of the current round.

Consider instead, with no loss of generality, that $p_i.qualifiers$ is a strict subset of $p_j.qualifiers$ in round r. As their values can only be binaries, in $\{0,1\}$, this means that $p_i.qualifiers$ is a singleton, say $\{w\}$. As all correct processes bv-deliver $r \bmod 2$ first, which is then broadcast into $p_i.favorites$, we have $w = r \bmod 2$ and p_i's estimate becomes $r \bmod 2$ at line 11. As $p_j.qualifiers$ is $\{0,1\}$, the estimate of p_j is also set to $r \bmod 2$ but at line 13. ◀

D Large TA

Table 3 details the rules for the first half of the threshold automaton from Fig. 3.

Table 3 The rules of the threshold automaton from Fig. 3. We omit self loops that have trivial guard *true* and no update.

Rules	Guard	Update
r_1	*true*	b_0++
r_2	*true*	b_1++
r_3	$b_0 \geq 2t{+}1{-}f$	a_0++
r_4	$b_1 \geq t{+}1{-}f$	b_1++
r_5	$b_0 \geq t{+}1{-}f$	b_0++
r_6	$b_1 \geq 2t{+}1{-}f$	a_1++
r_{14}, r_{15}, r_{16}	$a_0 \geq n{-}t{-}f$	—
r_8	$b_1 \geq t{+}1{-}f$	b_1++
r_9	$b_1 \geq 2t{+}1{-}f$	a_1++
r_{10}	$b_0 \geq 2t{+}1{-}f$	a_0++
r_{11}	$b_0 \geq t{+}1{-}f$	b_0++
r_{12}	$b_1 \geq 2t{+}1{-}f$	—
r_{13}	$b_0 \geq 2t{+}1{-}f$	—
r_7, r_{18}, r_{19}	$a_1 \geq n{-}t{-}f$	—
r_{16}	$a_0 \geq n{-}t{-}f$	—
r_{17}	$a_0{+}a_1 \geq n{-}t{-}f$	—
r_{20}, r_{21}, r_{22}	*true*	—

E Missing proof of Corollary 5

We restate here Corollary 5 and give its proof.

▶ **Corollary 9.** *Let $r \in \mathbb{N}$ be such that the rth execution of* bv-broadcast *in Algorithm 1 is* $(r \bmod 2)$*-good. Then:*

- *If there exists $R \in \mathbb{N}$ with $r = 2R{-}1$, then $\Box\left(\kappa[M_0, R] = 0\right)$ holds.*
- *If there exists $R \in \mathbb{N}$ with $r = 2R$, then $\Box\left(\kappa[M'_1, R] = 0\right)$ holds.*

Proof. By definition of an $(r \bmod 2)$-good execution, we know that in this particular invocation of bv-broadcast, all correct processes bv-deliver $r \bmod 2$ first. It follows from Lemma 4, that all correct processes start the next round with estimate set to $r \bmod 2$. There are two cases to consider depending on the parity of the round: If $r \bmod 2 = 1$, then this is the first round of superround R, i.e., $r = 2R - 1$. As a result, $\Box\left(\kappa[M_0, R] = 0\right)$. If $r \bmod 2 = 0$, then this is the second round of superround R, i.e., $r = 2R$. As a result, $\Box\left(\kappa[M'_1, R] = 0\right)$. ◀

F Specification of the termination property in the simplified threshold automaton for consensus algorithm

The reliable communication assumption and the properties guaranteed by the bv-broadcast are expressed as preconditions for *s_round_termination*. The progress conditions work exactly the same as in [10]. However, since the shared counters representing the bv-broadcast execution do not represent regular messages, we cannot directly use the reliable communication assumption. Instead, we use the properties of the bv-broadcast that we proved in a separate automaton.

In practice, instead of using progress preconditions on the bv-broadcast counters in *s_round_termination*, such as:

```
(locM  == 0 || bvb1 < 1) && (locM  == 0 || bvb0 < 1) &&
(locM1 == 0 || bvb0 < 1) && (locM0 == 0 || bvb1 < 1)
```

we use the following:

```
/* BV-Termination */
(locM    == 0) &&
/* BV-Obligation */
(locM1 == 0 || bvb0 < T + 1) && (locM0 == 0 || bvb1 < T + 1) &&
/* BV-Uniformity */
(locM1 == 0 || aux0 == 0) && (locM0 == 0 || aux1 == 0) &&
```

One can note that we do not use BV-Justification as a precondition in this specification. Instead, the BV-Justification property is baked in the structure of the simplified threshold automaton (in the guard of the transition $M \to M0, M1$).

The complete specification of the termination property follows:

```
s_round_termination:
<>[](
    (locV0   == 0) &&
    (locV1   == 0) &&

    /* BV-Termination */
    (locM    == 0) &&
    /* BV-Obligation */
    (locM1 == 0 || bvb0 < T + 1) &&
    (locM0 == 0 || bvb1 < T + 1) &&
    /* BV-Uniformity */
    (locM1 == 0 || aux0 == 0) &&
    (locM0 == 0 || aux1 == 0) &&

    /* Business as usual */
    (locM1   == 0 || aux1 < N - T) &&
    (locM0   == 0 || aux0 < N - T) &&
    (locM01  == 0 || aux0 + aux1 < N - T) &&

    (locD1   == 0) &&
    (locE0   == 0) &&
    (locE1   == 0) &&

    /* BV-Termination */
    (locMx    == 0 ) &&
    /* BV-Obligation */
    (locM1x    == 0 || bvb0x < T + 1) &&
    (locM0x    == 0 || bvb1x < T + 1) &&
    /* BV-Uniformity */
    (locM1x == 0 || aux0x == 0) &&
    (locM0x == 0 || aux1x == 0) &&

    (locM1x   == 0 || aux1x < N - T) &&
    (locM0x   == 0 || aux0x < N - T) &&
    (locM01x  == 0 || aux1x < N - T) &&
    (locM01x  == 0 || aux0x < N - T) &&
    (locM01x  == 0 || aux0x + aux1x < N - T)
    )
->
<>(
    locV0   == 0 &&
    locV1   == 0 &&
    locM    == 0 &&
    locM0   == 0 &&
    locM1   == 0 &&
    locM01  == 0 &&
    locE0   == 0 &&
    locE1   == 0 &&
    locD1   == 0 &&
    locMx   == 0 &&
    locM0x  == 0 &&
    locM1x  == 0 &&
    locM01x == 0
);

inv1_0: <>(locD0 != 0) -> [](locD1 == 0 && locE1x == 0);

inv2_0: [](locV0 == 0) -> [](locD0 == 0 && locE0x == 0);

inv1_1: <>(locD1 != 0) -> [](locD0 == 0 && locE0x == 0);

inv2_1: [](locV1 == 0) -> [](locD1 == 0 && locE1x == 0);

dec_0:  [](locV0 == 0) -> [](locE0 == 0 && locE1 == 0);

dec_1:  [](locV1 == 0) -> [](locE0x == 0 && locE1x == 0);

good_0: [](locM0 == 0) -> [](locD0 == 0 && locE0x == 0);

good_1: [](locM1x == 0) -> [](locE1x == 0);
```

How to Meet at a Node of Any Connected Graph

Subhash Bhagat
Département d'informatique, Université du Québec en Outaouais, Gatineau, Canada

Andrzej Pelc
Département d'informatique, Université du Québec en Outaouais, Gatineau, Canada

Abstract

Two mobile agents have to meet at the same node of a connected graph with unlabeled nodes. This intensely researched task is known as rendezvous. The adversary assigns the agents different starting nodes in the graph and different integer labels from a set $\{1, \ldots, L\}$. Time is slotted in synchronous rounds. The adversary wakes up the agents in possibly different rounds. After wakeup, the agents move as follows. In each round, an agent can either stay idle or move to an adjacent node. Each agent knows its label but not the label of the other agent, and agents have no a priori information about the graph. They do not know L. They execute the same deterministic algorithm whose parameter is the agent's label. The time of a rendezvous algorithm is the worst-case number of rounds since the wakeup of the earlier agent till the meeting.

In most of the results concerning rendezvous in graphs, the graph is finite and rendezvous relies on the exploration of the entire graph. Thus the time of rendezvous depends on the size of the graph. This approach is inefficient for very large graphs, and cannot be used for infinite graphs. For such graphs it is natural to seek rendezvous algorithms whose time depends on the initial distance D between the agents. In this paper we adopt this approach and consider rendezvous in arbitrary connected graphs with nodes of finite degrees, and whose set of nodes is finite or countably infinite. Our main result is the first deterministic rendezvous algorithm working under this general scenario.

For any node v and any positive integer r, let $P(v, r)$ be the number of paths of length r in the graph, starting at node v. For any instance of the rendezvous problem where agents start at nodes v_1 and v_2 at distance D, let $P(v_1, v_2, D) = \max(P(v_1, D), P(v_2, D))$. It is well known that, for example in trees, $\Omega(D + P(v_1, v_2, D) + \log L)$ is a lower bound on rendezvous time for such an instance. The time of our algorithm, working for arbitrary connected graphs of finite degrees, is polynomial in this lower bound.

As an application we solve the problem of approach for synchronous agents in terrains in the plane, in time polynomial in $\log L$ and in the initial distance between the agents in the terrain.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Distributed algorithms

Keywords and phrases Algorithm, graph, rendezvous, mobile agent, terrain

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.11

Funding *Andrzej Pelc*: Research supported by NSERC discovery grant 2018-03899 and by the Research Chair in Distributed Computing at the Université du Québec en Outaouais.

1 Introduction

Two mobile agents have to meet at the same node of a connected graph. This intensely researched task is known as *rendezvous* and has numerous applications. In computer networks, such as the internet, software agents navigate in the network and the purpose of meeting may be to share data collected from distributed databases and to plan further actions based on these data. If the network models a labyrinth, or corridors in a contaminated mine, mobile robots circulating in the network may have to meet to coordinate maintenance or decontamination tasks. Finally, people may want to meet in an unknown mall or park whose alleys are links of a network and crossings are its nodes.

© Subhash Bhagat and Andrzej Pelc;
licensed under Creative Commons License CC-BY 4.0

In most of the results concerning rendezvous in graphs, the graph is finite and rendezvous relies on the exploration of the entire graph. Thus the time of rendezvous depends on the size of the graph. This approach is inefficient for very large graphs, and cannot be used for infinite graphs. For such graphs it is natural to seek rendezvous algorithms whose time depends on the initial distance D between the agents. In this paper we adopt this approach and consider deterministic rendezvous in arbitrary connected graphs with nodes of finite degrees, and whose set of nodes is finite or countably infinite.

We also consider the problem of *approach* for synchronous[1] agents in terrains in the plane. This is the task of getting two agents with vision radius 1 navigating in a *terrain* (see the definition below) to see each other, i.e., to reach positions w_1 and w_2 in the terrain, such that the segment $[w_1, w_2]$ is of length at most 1 and is included in the terrain.

1.1 The model

Graphs. We consider arbitrary simple connected graphs with nodes of finite degrees, and whose set of nodes is finite or countably infinite. Nodes of the graph are unlabeled and ports at each node of degree d are arbitrarily labeled $0, 1, \dots, d-1$. There is no coherence between port numbering at different nodes. There are two agents to which the adversary assigns different starting nodes in the graph and different integer labels from a set $\{1, \dots, L\}$. Time is slotted in synchronous rounds. The adversary wakes up the agents in possibly different rounds. After wakeup, the agents move as follows. In each round, an agent can either stay idle or move to an adjacent node. An agent makes a move by choosing a port number at its current node. When entering the adjacent node corresponding to the chosen edge the agent learns the port of entry and the degree of this node. When agents cross each other in an edge, traversing it simultaneously in opposite directions, they do not even notice this fact. Agents cannot mark the visited nodes in any way. We assume that the memory of the agents is unbounded: from the computational point of view they are modeled as Turing machines. Each agent knows its label but not the label of the other agent, and agents have no a priori information about the graph. They do not know L. They execute the same deterministic algorithm whose parameter is the agent's label. The time of a rendezvous algorithm is the worst-case number of rounds since the wakeup of the earlier agent till the meeting. The meeting can occur before the wake-up of the later agent.

We will use the following terminology. A *walk* in a graph is any sequence $(v_0, v_1, \dots, v_k)$, such that $\{v_i, v_{i+1}\}$ is an edge, for any $i < k$. A *path* in a graph is any walk $(v_0, v_1, \dots, v_k)$ such that $v_{i+2} \neq v_i$ for any $i < k-1$. In other words, paths are walks with no immediate backtrack. Since an agent learns the entry port number upon visiting a node, it can avoid backtracking and thus it can travel only using paths.

Terrains. We consider mobile agents modeled by points moving in subsets of the Euclidean plane E^2 and equipped with compasses showing cardinal directions N, E, S, W, with a common unit of length, with clocks ticking at the same rate, and with vision of radius 1. This means that at any point u in the terrain, the agent sees all points v, such that the segment $[u, v]$ has length at most 1 and is included in the terrain.

Consider a finite or countably infinite family $\{O_1, O_2, \dots\}$ of pairwise disjoint closed convex subsets of the Euclidean plane E^2, called *obstacles*. For any real $\epsilon > 0$, such a family is called *ϵ-scattered* if all distances between obstacles are at least ϵ. A *terrain* is a subset of

[1] By this we mean that clocks of the agents tick at the same rate, and when they move, they travel at the fixed speed 1; see section 1.1.

the Euclidean plane which is the complement of the union of a ϵ-scattered family of obstacles, for some $\epsilon > 0$.[2] Hence any terrain is an open connected subset of the Euclidean plane. A terrain that is the complement of the union of a ϵ-scattered family of obstacles is itself called an ϵ-scattered terrain.

Agents wake up at distinct points p_1 and p_2 of the terrain at possibly different times, chosen by the adversary. The distance D in the terrain between points p_1 and p_2 is defined as the infimum of lengths of all polygonal lines between points p_1 and p_2, included in the terrain. The adversary also assigns different integer labels from a set $\{1, \ldots, L\}$ to the agents. The clock of every agent starts at its wake-up time. Each agent knows its own label but not the label of the other agent, and both of them know a common positive real $\epsilon \leq D$, such that the terrain is ϵ-scattered. As before, the memory of the agents is unlimited. Each agent executes a sequence of *actions*. An action can be either waiting at the current point for a chosen time t, or moving along a segment I of length $x \leq 1$ in a chosen direction dir, so that I is included in the terrain. Notice that, since the vision radius of the agents is 1, an agent can check the latter condition before the move. Whenever agents move, they move at the same fixed speed 1. Recall that they may start at different times and that one agent may move while the other waits. The approach is defined as the moment when the agents see each other for the first time. The time of an approach algorithm is the worst-case time since the wakeup of the earlier agent till the approach.

1.2 The lower bounds

We mention two well-known lower bounds on time, one for rendezvous in graphs and one for approach in the terrain.

Graphs. For any node v and any positive integer r, let $P(v, r)$ be the number of paths of length r in the graph, starting at node v. For any instance of the rendezvous problem where agents start at nodes v_1 and v_2 at distance D, let $P(v_1, v_2, D) = \max(P(v_1, D), P(v_2, D))$. It is well known that, for example in trees, $\Omega(D + P(v_1, v_2, D) + \log L)$ is a lower bound on rendezvous time for such an instance. Indeed, the lower bound $\Omega(D)$ is obvious, the lower bound $\Omega(\log L)$ follows from [14] (even in the two-node tree and even for simultaneous start) and the lower bound $\Omega(P(v_1, v_2, D))$ follows from the fact that the adversary can delay one of the agents and place it at the last node at distance D visited by the other agent.

Terrains. For any instance of the approach problem in a terrain T, with agents starting at points p_1, p_2, let D be the distance between p_1 and p_2 in T. Clearly, $\Omega(D)$ is a lower bound on the time of approach, as agents have speed 1. On the other hand, the lower bound $\Omega(\log L)$ holds even in the empty plane and follows from [14]. Hence we get the lower bound $\Omega(D + \log L)$.

1.3 Our results

Our main result is the solution of the feasibility problem of rendezvous in connected graphs under the above described general scenario. We design a rendezvous algorithm working for arbitrary connected graphs with nodes of finite degrees, and whose set of nodes is finite or countably infinite. Its execution time is polynomial in the above mentioned lower bound $\Omega(D + P(v_1, v_2, D) + \log L)$ for the rendezvous problem.

---

[2] Notice that obstacles do not need to be bounded. In general, disjoint closed unbounded sets in the plane may have distance 0 (such as a curve and its asymptote) but this possibility is precluded in terrains by the fact that the family of obstacles is ϵ-scattered.

This result should be compared to four sets of previous results about rendezvous in graphs, known in the literature. In [14, 19, 25], the authors considered rendezvous under the same scenario but only for finite graphs. The method adopted in these papers crucially relies on the finiteness of the graph, as it requires the exploration of the entire graph. Hence it cannot be applied to infinite graphs, and even in very large finite graphs it is inefficient. In [11], the authors considered some infinite graphs, such as the line and infinite multidimensional grids. They designed almost optimal rendezvous algorithms but they used two very strong assumptions: first, they assumed that the agents know their position in the graph, and second, they assumed simultaneous start. Hence, their results are far from our generality. In [7], the authors considered only trees (finite or infinite), and, in the unoriented case, only regular trees. The regularity assumption was important, as they relied on knowing the size of any ball of a given radius in the tree. (As explained in [7], the regularity assumption could be weakened to assuming that the size of any ball of given radius is bounded and that both agent know a common bound on this size). Hence again, while the complexity of algorithms in [7] is better than ours, their results are far from our generality. Finally, in [13], the authors designed a rendezvous algorithm working in arbitrary connected graphs (finite or infinite), but worked under the asynchronous scenario. In this scenario, meeting at a node cannot be guaranteed, and hence rendezvous conditions are relaxed to allow meeting inside an edge. Moreover, the algorithm from [13] is very inefficient, in particular, its worst-case cost is exponential in L.

To the best of our knowledge, we propose the first algorithm guaranteeing a meeting at a node in arbitrary connected graphs with nodes of finite degrees when each agent knows only its own label. This general result is possible by applying a new way of organizing activity and waiting periods of the agents. These periods are decided according to bits of (transformed) labels of agents. However, while in previous papers, activity meant either exploring the entire finite graph [25] or exploring a ball in infinite trees [7], in the present paper activity means traversing a single path. The second change consists in allocating rapidly increasing periods of time devoted to processing consecutive paths. The agent uses the time allocated to a given path π first traversing it, then waiting at the other end of it, and then traversing back the path π. So even in its activity period the agent spends a long time staying idle. These crucial changes (the "path-by-path" approach and long waiting periods at the end of each path) made it possible to get rid of the assumption of finiteness of the graph in [25] and of the regularity of the tree in [7].

As an application of this new method we solve the problem of approach for synchronous agents in terrains in the plane, in time polynomial in $\log L$ and in the initial distance D in the terrain between the agents. Hence in this scenario, the execution time of our algorithm is polynomial in the lower bound $\Omega(D + \log L)$ for the approach problem.

This result should be compared to five previous results about approach, known in the literature. In [11], the authors designed an almost optimal algorithm for approach but their algorithm had rather limited scope. First, it worked only for the obstacle-free plane, and second, it used two strong assumptions: that the agents know their position in the plane, and that they start simultaneously. In [13], the authors designed an algorithm for approach working for even more general subsets of the Euclidean plane than we do, and working for the asynchronous scenario but the worst-case cost of their algorithm was exponential in L. In [4], the authors designed an almost optimal algorithm for asynchronous approach in the obstacle-free plane, under a strong assumption that each agent knows its position in the plane. In [15], the authors designed a polynomial algorithm for approach of agents with possibly different steady speeds, but their algorithm worked only for the obstacle-free plane,

hence had significantly more limited scope. Finally, in [8], the authors strengthened the result from [15] by designing an algorithm for approach working for the asynchronous scenario at cost polynomial in $D + \log L$ but, again, their algorithm worked only for the obstacle-free plane.

1.4 Related Work

Results closest to the present paper were discussed in Section 1.3. In the present section we mention other related work. Rendezvous was studied both in the randomized and deterministic settings. An excellent survey of randomized rendezvous can be found in [2], cf. also [1, 5]. Deterministic rendezvous in networks is overviewed in [23, 24]. Rendezvous was also studied in geometric settings, such as the interval of the real line, e.g., [5, 6], and the plane, e.g., [3, 9, 12]. The task of meeting for more than two agents, called *gathering*, was investigated, e.g., in [16, 18, 20].

In the deterministic setting, investigations were mostly focused on the feasibility and time complexity of synchronous rendezvous in networks. In most of the literature concerning rendezvous in networks, nodes of the network are assumed to be unlabeled and marking nodes by agents is not allowed. In this case, anonymous agents cannot meet in many symmetric networks, e.g., in oriented rings, if they start simultaneously. The reason for this is the symmetry of the initial configuration. In order to make the task feasible, symmetry is usually broken by assigning the agents distinct labels and assuming that each agent knows only its own label. This is the same scenario as in the present paper and in the previously mentioned papers [7, 14, 19, 25]. Some authors studied a weaker scenario in which agents, as well as nodes, are anonymous. Gathering many anonymous agents in unlabeled networks was the subject of [16]. In this weak scenario, not all initial configurations of agents are possible to gather, and the authors of [16] characterized all such configurations. Stronger scenarios were also investigated. The authors of [22] studied the time of rendezvous in labeled networks, in the context of algorithms with advice.

In the asynchronous model, an adversary controls the speed of agents. Asynchronous rendezvous and approach in the plane was studied in [8, 10, 18], and asynchronous rendezvous in graphs was introduced in [21] and later investigated in [4, 17].

2 Rendezvous in connected graphs

2.1 The algorithm

We first introduce some notation and terminology.

For any label $\ell \in \{1, \ldots, L\}$ we define the *transformed label* $Trans(\ell)$ as follows. Let $(c_1, \ldots, c_s)$ be the binary representation of ℓ (with $c_1 = 1$). First we define the binary sequence $Trans_1(\ell)$ as follows. We replace each bit 1 by 10, each bit 0 by 01 and add bits 11 at the end. The obtained sequence is of length $2s + 2$ and has the property that if we start with two different labels then none of the obtained sequences can be a prefix of the other (cf. [14]). In order to get $Trans_2(\ell)$ we replace in $Trans_1(\ell)$ each bit 1 by 10 and each bit 0 by 01. The resulting sequence $Trans_2(\ell)$ has length $4s + 4$. Notice that since binary representations of labels may have different lengths, the same is true for the resulting sequences $Trans_2(\ell)$. However, they have the property that if $\ell_1 \neq \ell_2$ then there exists an index j, such that the jth bit of $Trans_2(\ell_1)$ is 1 and the jth bit of $Trans_2(\ell_2)$ is 0. (Hence, not only $Trans_2(\ell_1)$ and $Trans_2(\ell_2)$ differ in some bit but we can guarantee that in some position the bits are 1 and 0 and in some other position the bits are 0 and 1). Finally, we

obtain $Trans(\ell)$ from $Trans_2(\ell)$ by adding to it the prefix 011. The resulting sequence of length $4s + 7$ still has the previous two properties (being prefix-free and guaranteeing that if $\ell_1 \neq \ell_2$ then there exists an index j, such that the jth bit of $Trans_2(\ell_1)$ is 1 and the jth bit of $Trans_2(\ell_2)$ is 0) and moreover it has the third property that the segment consisting of bits with indices 2,3,4 is the only segment of three consecutive bits 1 in $Trans(\ell)$.

As an example consider label $\ell = 9$. Then the binary representation of ℓ is 1001. We have $Trans_1(\ell) = (1001011011)$, $Trans_2(\ell) = (10010110011010011010)$, and $Trans(\ell) = (01110010110011010011010)$.

We define the infinite binary sequence $Tape(\ell)$ as the concatenation of infinitely many copies of $Trans(\ell)$. We will call $Tape(\ell)$ the tape of the agent with label ℓ. The i-th copy of $Trans(\ell)$ is called the i-th segment of $Tape(\ell)$.

Any path $\pi = (v_0, v_1, \dots, v_k)$ of length k starting at node v_0 is coded as the sequence of port numbers $(p_0, \dots, p_{k-1})$ such that port p_i at node v_i leads to node v_{i+1}. We will often identify a path with its code. The length k of path π is denoted by $|\pi|$. We denote by $rev(\pi)$ the reverse path $(v_k, v_{k-1}, \dots, v_0)$ coded by the sequence of port numbers $(q_k, q_{k-1}, \dots, q_1)$, such that port q_j at node v_j leads to node v_{j-1}. Since an agent learns the entry port upon entering a node, an agent that has traversed the path π learns (the code of) the path $rev(\pi)$.

For any node v, we define the infinite sequence of all finite paths $(\pi_1, \pi_2, \dots)$ starting at node v and ordered as follows: every path of smaller length precedes every path of larger length, and paths of a given length are ordered lexicographically by their codes. Since in our algorithm all paths of smaller lengths are traversed by the agents before all paths of larger length, when an agent has finished processing all paths of length i, it knows (the codes of) all paths of length $i + 1$ because when an agent is at the end of a path of length i, it sees the degree of the final node.

The high-level idea of the Algorithm RV, guaranteeing rendezvous in any connected graph with nodes of finite degrees, is the following. The algorithm is executed by an agent with label ℓ starting at a node v. We assign rapidly increasing time periods a_i (in our solution, the integers a_i increase quadratically) to process consecutive bits b_i of $Tape(\ell)$ of the agent. If the bit b_i is in the j-th segment of $Tape(\ell)$, then its processing concerns the path π_j starting at v, in the following way:

- if $b_i = 1$ then the agent traverses path π_j, waits $a_i - 2|\pi_j|$ rounds at the end of it, and traverses path $rev(\pi_j)$;
- if $b_i = 0$ then the agent waits a_i rounds.

Note that the agent starts and ends processing each bit of $Tape(\ell)$ at its starting node v. We will show that rendezvous must occur by the time when one of the agents processes all bits of its $Tape(\ell)$ corresponding to the lexicographically smallest among shortest paths from its initial position to the initial position of the other agent.

Below we give the pseudo-code of the algorithm. For any positive integer i we define $a_i = 3i^2$. The algorithm is interrupted as soon as the agents meet.

▶ Remark. To show that the formulation of the algorithm is correct, we need to prove that if b_i is the i-th bit of $Tape(\ell)$ located in the j-th segment of $Tape(\ell)$ then $a_i \geq 2|\pi_j|$. Indeed, we have $a_i = 3i^2 \geq 2i \geq 2j \geq 2|\pi_j|$.

2.2 Correctness and complexity

In this section we prove the correctness of Algorithm RV and establish its complexity. For an instance of the rendezvous problem, where agents A_1 and A_2 start at nodes v_1 and v_2 respectively, we define the critical segment of A_1 as follows. Let π be the lexicographically

Algorithm 1 Algorithm RV.

```
for i = 1, 2, ... do
    if b_i is the i-th bit of Tape(ℓ) located in the j-th segment of Tape(ℓ) then
        if b_i = 1 then
            traverse path π_j;
            wait a_i − 2|π_j| rounds;
            traverse path rev(π_j);
        else
            wait a_i rounds;
```

smallest among shortest paths from v_1 to v_2. We call path π the critical path of the agent. The critical segment of the tape of agent A_1 is the segment of its tape assigned to path π. The critical segment of the tape of agent A_2 is defined similarly.

The correctness of Algorithm RV follows from the two following lemmas.

▶ **Lemma 1.** *Suppose that the agents start executing Algorithm* RV *simultaneously. Then they meet by the end of the execution of the critical segment of the agent that starts its critical segment first.*

Proof. Let A_1 be the agent that starts its critical segment earlier and let A_2 be the other agent. If both agents start their critical segments simultaneously, we call A_1 and A_2 arbitrarily. Let T be the round in which A_1 starts its critical segment. Consider two cases.

Case 1. Agent A_2 starts some segment in round T. For $i = 1, 2$, let σ_i be the segment that agent A_i starts in round T. There exists an index j such that the j-th bit of agent A_1 is 1, the j-th bit of agent A_2 is 0, and these bits are in segments σ_1 and σ_2, respectively. Hence, in the first part of the execution of its j-th bit, agent A_1 traverses its critical path, while agent A_2 waits at its other end. Thus the agents meet at the starting node of A_2.

Case 2. Agent A_2 does not start any segment in round T. Let σ_2 be the segment which agent A_2 is processing in round T. Let (d_1, d_2, d_3, d_4) be the four bits that agent A_2 executes starting in round T, i.e., while agent A_1 executes the first four bits of its critical segment. (d_1, d_2, d_3, d_4) are not the first four bits of the segment σ_2 of agent A_2. If all bits d_2, d_3, d_4 are within segment σ_2 then at least one of them must be 0, as there cannot be three consecutive bits 1 (apart from positions 2,3,4 in a segment). If some of these bits are within the segment τ following σ_2, then one of them must be 0, since every segment starts with a 0. Hence, in any case, there exists an index j such that the j-th bit of agent A_1 is 1 and the j-th bit of agent A_2 is 0. Hence, in the first part of the execution of its j-th bit, agent A_1 traverses its critical path, while agent A_2 waits at its other end. Thus the agents meet at the starting node of A_2. ◀

▶ **Lemma 2.** *Suppose that the agents do not start executing Algorithm* RV *simultaneously. Then they meet by the end of the execution of the critical segment of the agent that starts executing Algorithm* RV *earlier.*

Proof. Let A_1 be the agent that starts the execution of Algorithm RV earlier, and let A_2 be the other agent. Let T be the round in which A_1 starts its critical segment and let $k-1$ be the index of the first bit in this segment. Hence agent A_1 starts the execution of the second bit of its critical segment (which is equal to 1) in round $T' = T + a_{k-1}$. Let δ be the delay in the start of execution of agent A_2 w.r.t. the start of agent A_1. We have following two cases.

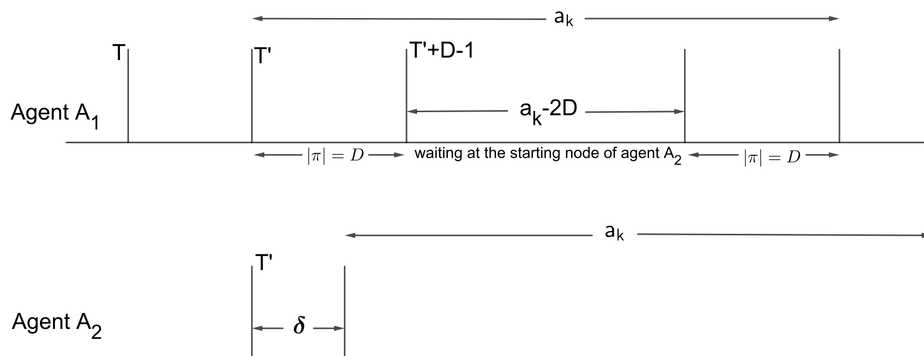


Figure 1 An illustration for the proof of Lemma 2: when $\delta \leq D$, agent A_1 meets agent A_2 during the execution of the k-th bit of agent A_1.

- $\delta \leq D$ **:** Let us consider the time interval $[T', T' + D - 1]$ (cf. Fig. 1). During this time interval agent A_1 traverses its critical path π (of length D) by processing its k-th bit. Since $\delta \leq D$, the execution of the $k - th$ bit of agent A_2 starts in the time interval $[T', T' + D - 1]$. If the k-th bit of A_2 is 0, then we are done, because agent A_1 meets agent A_2 at the starting node of A_2 during the execution of this bit of agent A_2. Otherwise, the k-th bit of A_2 is 1 and we have the following two possibilities. If the k-th bit of A_2 is not the first bit 1 of some segment σ_2, then at least one among the $(k+1)$-th and $(k+2)$-th bits of A_2 is 0. Since the k-th, the $(k+1)$-th and the $(k+2)$-th bits of A_1 are 1, the agents meet at the starting node of A_2 during the execution of the first bit 0 of A_2 following its k-th bit. Now consider the case when the k-th bit of A_2 is the first bit 1 of some segment σ_2. The start of the execution of segment σ_2 by agent A_2 is delayed by at most D with respect to the start of the execution of the critical segment by agent A_1. Hence, for some index m, the m-th bit of agent A_1 is 1, the m-th bit of agent A_2 is 0 and the delay between the executions of these bits is most D. Hence the agents meet at the starting node of A_2, during the executions of their m-th bits.

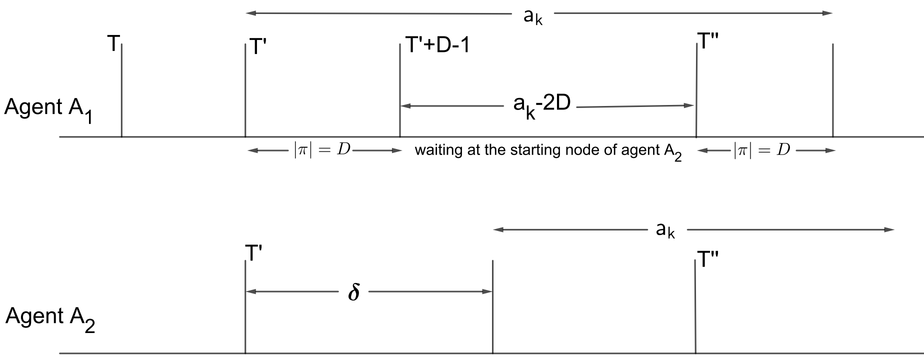


Figure 2 An illustration for the proof of Lemma 2: when $\delta > D$ and $\delta \leq T'' - T'$, agent A_1 meets agent A_2 during the execution of the k-th bit of agent A_1.

- $\delta > D$ **:** Let j be the index of the bit which agent A_2 is processing in round $T' + D$. Since $\delta > D$, we have $j < k$.
  Consider the processing of the k-th bit of agent A_1. Since this bit is 1, agent A_1 first traverses its critical path π (of length D), then waits $a_k - 2D$ rounds at the other end of π (which is the starting node of A_2), and finally traverses the reverse path $rev(\pi)$. We will use the following claim.

  ▷ Claim. $a_k - 2D > a_j$.

  To prove the claim notice that $a_j \leq a_{k-1}$, since $j < k$. Since $1 \leq D \leq k$, we have

$$a_k - a_j \geq a_k - a_{k-1} = 3k^2 - 3(k-1)^2 = 6k - 3 \geq 6D - 3 > 2D,$$

  which proves the claim.
  Let $T'' = T' + a_k - D - 1$. Consider the time interval $I = [T' + D, T'']$ which is the time interval during which agent A_1 waits at the starting node of A_2. If $\delta \leq T'' - T'$ then the start of the execution of the k-th bit of A_2 happens during the time interval I, hence the agents meet at the starting node of A_2 (cf. Fig. 2). If $\delta > T'' - T'$ then during the last round of interval I agent A_2 executes some j-th bit, for $j < k$. By the claim, the start of the execution of the j-th bit of agent A_2 happens during the waiting period of agent A_1 (cf. Fig. 3). Hence the agents meet at the starting node of A_2 during time interval I.

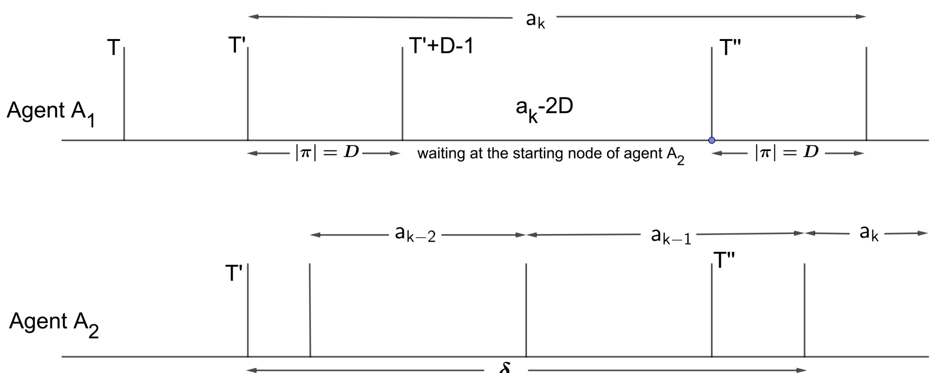


Figure 3 An illustration for the proof of Lemma 2: when $\delta > D$ and $\delta > T'' - T'$, agent A_1 meets agent A_2 during the execution of the k-th bit of agent A_1. ◀

We are now ready to prove the main result of this section.

▶ **Theorem 3.** *Algorithm* RV *guarantees rendezvous of agents starting at nodes v_1 and v_2 at distance D in an arbitrary connected graph (finite or infinite) in time polynomial in $D + P(v_1, v_2, D) + \log L$.*

Proof. Let A be the agent that started the execution of the algorithm first, and in the case of simultaneous start, let it be the agent that started its critical segment first. Let v be the starting node of A. By Lemmas 1 and 2 we know that the agents will meet by the end of the execution of the critical segment of agent A. Let P be the number of paths of length at most D starting at v. We have

$$P = \sum_{i=1}^{D} P(v,i) \leq D \cdot P(v,D).$$

The number of segments till the critical segment of A is at most P. Each segment contains at most $c \log L$ bits, for some constant c. Hence the number B of all bits until the end of the critical segment is at most $cP \log L$. The execution time of the algorithm is at most

$$\sum_{j=1}^{B} a_j = \sum_{j=1}^{B} 3j^2 = \frac{B(B+1)(2B+1)}{2} \in O(B^3).$$

On the other hand we have

$$O(B^3) \subseteq O((P \log L)^3) \subseteq O((D \cdot P(v,D) \cdot \log L)^3) \subseteq O((D \cdot P(v_1,v_2,D) \cdot \log L)^3),$$

which is polynomial in $D + P(v_1, v_2, D) + \log L$. ◀

3 Approach in terrains

In this section we use the path-by-path method from Algorithm RV to solve the problem of approach for synchronous agents in terrains, in time polynomial in $\log L$ and in the initial distance D between the agents in the terrain. We do it in two steps. First, we modify Algorithm RV to obtain an efficient rendezvous algorithm for arbitrary connected subgraphs of the infinite oriented grid, and then we derive an algorithm for approach from this modified algorithm.

We consider the infinite oriented grid $\mathbb{Z} \times \mathbb{Z}$. Every node is adjacent to the four nodes at distance 1 from it in directions North, East, South, West. Ports at nodes of the grid are denoted N, E, S, W, according to the orientation. We define a *shape* as any connected subgraph of this grid. Mobile agents navigate in a fixed shape along its edges. An agent located at a current node of the shape knows which of the ports correspond to edges in the shape. The agents have the same characteristics as described for general connected graphs. Our first aim is to design a rendezvous algorithm working in arbitrary shapes in time polynomial in $\log L$ and in the initial distance D between the agents in the shape.

If we were not concerned with the efficiency, we could directly use Algorithm RV, as shapes are connected graphs. However, this would not give us the desired complexity polynomial in $\log L$ and in D. Indeed, recall that Algorithm RV guarantees rendezvous of agents starting at nodes v_1 and v_2 at distance D in time polynomial in $P(v_1, v_2, D) + \log L$. For shapes (in fact even for the empty grid) the number of paths of length D between two nodes at distance D may be exponential in D. (For example, the number of paths of length $2a$ between nodes (x, y) and $(x + a, y + a)$ which are at distance $2a$ in the grid is $\binom{2a}{a}$).

Luckily, we can significantly reduce the number of processed paths using the orientation of the grid. As in Algorithm RV, paths in shapes are ordered so that every path of smaller length precedes every path of larger length, and paths of a given length are ordered lexicographically by their codes which are sequences of ports N, E, S, W ordered $N < E < S < W$. The key change is to avoid processing all paths in the shape. We do it by associating to each node w in the shape at distance i from the starting node v of the agent, a single specific path of length i called *canonical*. This is the lexicographically smallest of all paths of length i from v to w in the shape. Since all paths of smaller lengths are traversed by the agents before all paths of larger length, when an agent has finished processing all canonical paths of length i, it knows (the codes of) all canonical paths of length $i + 1$. Indeed, the canonical path

of length $i+1$ from v to w has a prefix of length i which is the canonical path from v to a neighbor w' of w at distance i from v in the shape. Thus the agent can determine this canonical path to w before starting its traversal.

This is the only change we make in Algorithm `RV`: the sequence $(\pi_1, \pi_2, \dots)$ is now the sequence of all canonical paths in the shape, starting at v, ordered as above, and the rest of the algorithm is as before.[3] The resulting rendezvous algorithm, working for arbitrary shapes, is called Algorithm `Shape-RV`. Let $C(v,i)$ be the number of canonical paths of length i in the shape, starting at node v. By definition, $C(v,i)$ is equal to the number of nodes in the shape at distance i from v, and since shapes are subgraphs of the grid, we have that $C(v,i)$ is in $O(i^2)$. The same analysis as for Algorithm `RV` proves the following lemma.

▶ **Lemma 4.** *Algorithm* `RV-Shape` *guarantees rendezvous of agents starting at nodes v_1 and v_2 at distance D in an arbitrary shape in time polynomial in $D + \log L$. Rendezvous occurs at the starting node of one of the agents.*

We are now ready to make the second step in our design of the algorithm for approach in terrains. Let us first consider agents operating in the same shape, as above, but in a slightly changed model. Instead of operating in synchronous rounds (as is usually the case for the graph setting) we allow the agents to start with any positive delay δ, not necessarily integer. This means that the later agent may start while the earlier agent is traversing an edge. We use the same algorithm as above, i.e, Algorithm `RV-Shape`. It is easy to see that Lemma 4 still holds in this slightly more general situation. Indeed, the proof of Lemma 1 does not need any change (since it deals with the case $\delta = 0$) and the proof of Lemma 2 requires only minimal changes, replacing rounds by points in time. (For example, the time interval $[T', T' + D - 1]$ which meant a segment of D rounds in the proof of Lemma 2 would have to be replaced by the time interval $[T', T' + D]$ meant as a time segment of length D between two points in time).

It is well known [13, 15] that the problem of approach in the empty plane (without obstacles) can be reduced to that of rendezvous in an infinite oriented grid. For completeness we sketch this reduction below. For any point v in the plane, consider the infinite oriented grid G_v defined as the following graph embedded in the plane. One of the nodes of G_v is v and every node u is adjacent to 4 nodes at Euclidean distance 1 from it, and located North, East, South and West from node u. Ports at every node are labeled N, E, S, W, in the obvious way.

Any rendezvous algorithm in the grid G_v (whose aim is to bring two agents starting at arbitrary nodes of the grid with arbitrary delay to the same node at the same time) can be transformed in an approach algorithm in the empty plane as follows. Let A be any rendezvous algorithm for G_v. Algorithm A can be executed in the grid G_w, for any point w in the plane. Consider two agents in the plane starting respectively from point v and from another point w in the plane, with some delay δ. Let v' be the node of G_v closest to point w. We will say that v and v' are *companions*. If there are more than one closest nodes, we pick one of them arbitrarily. Notice that v' is at distance at most $\sqrt{2}/2 < 1$ from w. Let α be the vector $v'w$. Execute algorithm A on the grid G_v with agents starting at nodes v and v' with delay δ. Let u be the node of G_v in which these agents meet at some time t. The transformed algorithm A^* for approach in the plane works as follows: execute the

[3] Notice that we could not apply this method for arbitrary connected graphs. In an anonymous graph it is impossible to tell if two paths with given codes end up at the same node or not. This shows the power of the grid orientation.

same algorithm A but with one agent starting at v and traveling in G_v and the other agent starting at w and traveling in G_w, so that the starting time of the agent starting at w is the same as the starting time of the agent starting at v' in the execution of A in G_v; the starting time of the agent starting at v does not change. In time t the agent starting at v and traveling in G_v will be at point p, as previously. The agent starting at w and traveling in G_w will get to some point q at time t. Clearly, p is a node of G_v, q is a node of G_w, points p and q are companions and $q = p + \alpha$. Hence both agents will be at distance less than 1 at time t, which means that they accomplish approach in the (empty) plane. Notice that the transformed algorithm A^* has the same time as algorithm A.

Unfortunately, the above transformation does not guarantee approach even if there is one closed convex obstacle in the plane. Indeed, this obstacle could be positioned in such a way that the segment $[p, q]$ intersects it, and hence although agents get close to each other, they cannot see each other. This is why we need a preprocessing for the approach algorithm and we need to carefully choose the length of edges of the grids depending on the given parameter ϵ, such that the terrain is ϵ-scattered.

We will use the following geometric observation.

▶ **Lemma 5.** *Let v be any point in an ϵ-scattered terrain. Let $\xi = \min(1, \epsilon)$. Then there exists a point v' in the terrain at distance $\xi/3$ from v, such that the distance between v' and any obstacle is at least $\xi/3$.*

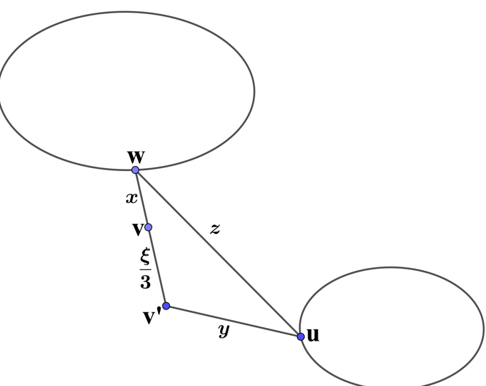


Figure 4 An illustration for the proof of Lemma 5.

Proof. If the distance between v and any obstacle is at least $\xi/3$ then we can take $v' = v$. Hence suppose that the distance between v and the closest obstacle O is $x < \xi/3$. Let w be the point in O such that the distance between v and w is x (cf. Fig. 4).

Let v' be the (unique) point in the terrain in the line vw at distance $\xi/3$ from v. (The existence of such a point follows from the fact that the terrain is ϵ-scattered, and the unicity follows from the definition of x.) Clearly, v' is at distance larger than $\xi/3$ from O. Consider any other obstacle O'. It is enough to show that v' is at distance at least $\xi/3$ from O'. Let u be the point of O' closest to v' and let y be the distance between u and v'. Let the distance between w and u be z. Consider the triangle $wv'u$. We have $x + \xi/3 + y \geq z \geq \epsilon \geq \xi$. Hence $y \geq \xi/3$. ◀

The preprocessing part of our algorithm for approach executed by agent A is the following Procedure `Away`(ϵ) which takes as parameter a positive real ϵ such that the terrain is ϵ-scattered. Recall that both agents get the same ϵ as input. Procedure `Away`(ϵ) works as

follows. Let $\xi = \min(1, \epsilon)$ and let $\lambda = \xi/3$. Let v be the starting point of agent A. If the agent does not see any obstacles at distance less than λ then it defines $v' = v$ and terminates the procedure. Otherwise, the agent defines w to be the closest point in the closest obstacle. Then it goes along the line vw away from the point w to the point v' at distance λ from v. Since the terrain is ϵ-scattered, the point v' is in the terrain. This concludes the procedure. By Lemma 5, point v' is at distance at least λ from any obstacle.

Consider the grid $H_{v'}$ which is defined similarly as the above grid $G_{v'}$, with the only exception that adjacent nodes are at distance λ instead of distance 1. Define the shape $S_{v'}$ as the subgraph of $H_{v'}$ induced by nodes that are points of the terrain. Now the Algorithm `Approach` executed by agent A can be succinctly described as follows. Execute Procedure `Away`(ϵ) to get to point v' and then execute Algorithm `RV-Shape` in the shape $S_{v'}$.

The following theorem proves the correctness and establishes the complexity of Algorithm `Approach`.

▶ **Theorem 6.** *Algorithm* `Approach` *accomplishes approach of arbitrary agents in any ϵ-scattered terrain in time polynomial in $D + \log L$, where D is the distance between the starting points of the agents in the terrain.*

Proof. Consider two agents A and B, starting from points v and w respectively. Let v' and w' be the points which the agents A and B reach after executing the Procedure `Away`. Algorithm `RV-Shape` is now executed by agent A in shape $S_{v'}$ and by agent B in shape $S_{w'}$. First suppose that $S_{v'} = S_{w'}$. In this case Lemma 4 guarantees that agents will meet either at v' or at w'. Without loss of generality, suppose that they meet at w'. Now suppose that $S_{v'} \neq S_{w'}$. Hence the agents operate in different shapes. Let w^* be the companion node of w'. The point w^* is a node in the shape $S_{v'}$. Lemmas 1 and 2 can still be used to guarantee that when agent A gets to node w^* during the execution of its critical segment (where the critical path is now from w' to w^*) then agent B is at point w'. This is because the considerations in the proofs of Lemmas 1 and 2 do not depend on which paths agent B is traversing in the execution of Algorithm `RV-Shape` (these paths may depend on the shape in which it operates) but only on the waiting times at its starting node which are independent of the shape.

In view of Lemma 5, there is no obstacle at distance less than λ from point w'. The companion w^* of w' is at distance smaller than λ from w'. Hence the segment w^*w' is contained in the terrain and has length smaller than 1, and thus the agents can see each other. This proves the correctness of Algorithm `Approach`.

As for complexity, first observe that the duration of Procedure `Away` is at most $\lambda \leq D$ (because agents were given a common $\epsilon \leq D$ such that the terrain is ϵ-scattered, and $\lambda \leq \epsilon$). Notice that since the shape $S_{v'}$ is the subgraph of $H_{v'}$ induced by nodes that are points of the terrain, the distance between any two nodes of the shape (defined as the distance in the graph) is at most twice the distance D between these points in the terrain. In view of Lemma 4, we conclude that the execution time of Algorithm `RV-Shape` (which is the second part of Algorithm `Approach`) is polynomial in $D + \log L$. Hence the execution time of the entire Algorithm `Approach` is also polynomial in $D + \log L$. ◀

4 Conclusion

We presented two deterministic algorithms: one for rendezvous in arbitrary connected graphs with nodes of finite degrees (whose set of nodes is finite or infinite) and the other for approach in terrains in the plane. For the rendezvous algorithm the scope is the most general possible. Indeed, rendezvous is obviously impossible in disconnected graphs, and if a graph has a

node of infinite degree then there is no rendezvous algorithm guaranteed to always finish in any finite time T, even if agents start from adjacent nodes. The remaining open problem concerns complexity. Our algorithm works in time polynomial in $D + P(v_1, v_2, D) + \log L$ and we did not try to optimize the degree of this polynomial. The problem of designing a rendezvous algorithm working for arbitrary connected graphs in optimal time remains open. This is a challenging problem: it remains unsolved even for finite graphs, despite two decades of intense research.

It is important to note that techniques used in the literature for rendezvous in finite graphs do not seem to have an easy extension to the case of infinite graphs. For example, the idea of trying increasing guesses on the size n of the graph and running known algorithms for finite graphs does not seem to work in infinite graphs for the following reason. Rendezvous algorithms for graphs with size bounded by n rely on exploration of the entire graph using sequences $UXS(n)$ (i.e., Universal Exploration Sequences of port numbers for graphs of size at most n), cf. [25]. Such a sequence guarantees visiting all nodes of a graph of size at most n but does not give any guarantee of visiting some ball in an infinite graph. This is due to the anonymity of the graphs: an agent cannot detect if it enters a loop instead of visiting all nodes at distance r. Existing rendezvous algorithms for finite graphs rely on one agent waiting at its starting node and the other catching it by visiting this node, which could not be guaranteed for any guess. Another possibility would be to exhaustively explore a ball of a given radius r (for increasing values of r) by traversing all paths of length r (coded as sequences of port numbers) starting at the starting node of the agent. However, in arbitrary infinite graphs, there is no upper bound on the number of such paths, and thus it would be impossible to determine a sufficiently long waiting period for agent A at its starting node to guarantee that the other agent B certainly catches it. These difficulties forced us to invent the "path-by-path" technique used in this paper.

For approach, three problems remain open: the first concerns the scope, the second concerns information available to the agents, and the third concerns complexity. Our algorithm works for ϵ-scattered terrains and its time is polynomial in $D + \log L$, while $\Omega(D + \log L)$ is the lower bound on the complexity of approach. This invites three questions. The first concerns the generality of the environment. While our class of terrains is fairly large, it is natural to ask if there exists an algorithm for approach working in arbitrary open connected subsets of the plane, with similar complexity. An algorithm for approach working in all such subsets could be easily obtained by a modification of the result from [13] but its complexity is prohibitive: it is exponential in L. The second problem concerns the information available to the agents. We assumed that each agent knows its own label, and both of them know a common real $\epsilon \leq D$, such that the terrain is ϵ-scattered. The first assumption is necessary to break symmetry: anonymous agents walking at the same speed cannot meet deterministically even in the empty plane, if they start simultaneously. However, we may ask if agents can meet in ϵ-scattered terrains without any other knowledge, with complexity similar to that of our algorithm. The third problem concerns optimal complexity of approach. What is the optimal time of approach even only in our setting of ϵ-scattered terrains? This is not known even for the plane without any obstacles. In this case, the best known algorithm has time $O(D^2 \log L)$ (folklore) and the best known lower bound is $\Omega(D^2 + D \log L)$ and follows from [14].

References

1 Steve Alpern. The rendezvous search problem. *SIAM Journal on Control and Optimization*, 33(3):673–683, 1995. doi:10.1137/S0363012993249195.

2 Steve Alpern and Shmuel Gal. *The theory of search games and rendezvous*, volume 55 of *International series in operations research and management science*. Kluwer, 2003.

3 Edward J. Anderson and Sándor P. Fekete. Two dimensional rendezvous search. *Operations Research*, 49(1):107–118, 2001. doi:10.1287/opre.49.1.107.11191.

4 Evangelos Bampas, Jurek Czyzowicz, Leszek Gasieniec, David Ilcinkas, and Arnaud Labourel. Almost optimal asynchronous rendezvous in infinite multidimensional grids. In *Proc. 24th International Symposium on Distributed Computing (DISC 2010)*, volume 6343, pages 297–311. Springer, 2010. doi:10.1007/978-3-642-15763-9_28.

5 Vic Baston and Shmuel Gal. Rendezvous on the line when the players' initial distance is given by an unknown probability distribution. *SIAM Journal on Control and Optimization*, 36(6):1880–1889, 1998. doi:10.1137/S0363012996314130.

6 Vic Baston and Shmuel Gal. Rendezvous search when marks are left at the starting points. *Naval Research Logistics (NRL)*, 48(8):722–731, 2001. doi:10.1002/nav.1044.

7 Subhash Bhagat and Andrzej Pelc. Deterministic rendezvous in infinite trees. *CoRR*, abs/2203.05160, 2022. doi:10.48550/arXiv.2203.05160.

8 Sébastien Bouchard, Marjorie Bournat, Yoann Dieudonné, Swan Dubois, and Franck Petit. Asynchronous approach in the plane: a deterministic polynomial algorithm. *Distributed Computing*, 32(4):317–337, 2019. doi:10.1007/s00446-018-0338-2.

9 Sébastien Bouchard, Yoann Dieudonné, Andrzej Pelc, and Franck Petit. Almost universal anonymous rendezvous in the plane. In *Proc. 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2020)*, pages 117–127. ACM, 2020. doi:10.1145/3350755.3400283.

10 Mark Cieliebak, Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Distributed computing by mobile robots: Gathering. *SIAM Journal on Computing*, 41(4):829–879, 2012. doi:10.1137/100796534.

11 Andrew Collins, Jurek Czyzowicz, Leszek Gasieniec, Adrian Kosowski, and Russell A. Martin. Synchronous rendezvous for location-aware agents. In *Proc. 25th International Symposium on Distributed Computing (DISC 2011)*, volume 6950, pages 447–459. Springer, 2011. doi:10.1007/978-3-642-24100-0_42.

12 Jurek Czyzowicz, Leszek Gasieniec, Ryan Killick, and Evangelos Kranakis. Symmetry breaking in the plane: Rendezvous by robots with unknown attributes. In *Proc. 2019 ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 4–13. ACM, 2019. doi:10.1145/3293611.3331608.

13 Jurek Czyzowicz, Andrzej Pelc, and Arnaud Labourel. How to meet asynchronously (almost) everywhere. *ACM Trans. Algorithms*, 8(4):37:1–37:14, 2012. doi:10.1145/2344422.2344427.

14 Anders Dessmark, Pierre Fraigniaud, Dariusz R. Kowalski, and Andrzej Pelc. Deterministic rendezvous in graphs. *Algorithmica*, 46(1):69–96, 2006. doi:10.1007/s00453-006-0074-2.

15 Yoann Dieudonné and Andrzej Pelc. Deterministic polynomial approach in the plane. *Distributed Computing*, 28(2):111–129, 2015. doi:10.1007/s00446-014-0216-5.

16 Yoann Dieudonné and Andrzej Pelc. Anonymous meeting in networks. *Algorithmica*, 74(2):908–946, 2016. doi:10.1007/s00453-015-9982-0.

17 Yoann Dieudonné, Andrzej Pelc, and Vincent Villain. How to meet asynchronously at polynomial cost. *SIAM Journal on Computing*, 44(3):844–867, 2015. doi:10.1137/130931990.

18 Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Peter Widmayer. Gathering of asynchronous robots with limited visibility. *Theor. Comput. Sci.*, 337(1-3):147–168, 2005. doi:10.1016/j.tcs.2005.01.001.

19 Dariusz R. Kowalski and Adam Malinowski. How to meet in anonymous network. *Theor. Comput. Sci.*, 399(1-2):141–156, 2008. doi:10.1016/j.tcs.2008.02.010.

20 Wei Shi Lim and Steve Alpern. Minimax rendezvous on the line. *SIAM Journal on Control and Optimization*, 34(5):1650–1665, 1996. doi:10.1137/S036301299427816X.

21 Gianluca De Marco, Luisa Gargano, Evangelos Kranakis, Danny Krizanc, Andrzej Pelc, and Ugo Vaccaro. Asynchronous deterministic rendezvous in graphs. *Theor. Comput. Sci.*, 355(3):315–326, 2006. doi:10.1016/j.tcs.2005.12.016.

22 Avery Miller and Andrzej Pelc. Tradeoffs between cost and information for rendezvous and treasure hunt. *J. Parallel Distributed Comput.*, 83:159–167, 2015. doi:10.1016/j.jpdc.2015.06.004.

23 Andrzej Pelc. Deterministic rendezvous in networks: A comprehensive survey. *Networks*, 59(3):331–347, 2012. doi:10.1002/net.21453.

24 Andrzej Pelc. Deterministic rendezvous algorithms. In Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors, *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, volume 11340 of *Lecture Notes in Computer Science*, pages 423–454. Springer, 2019. doi:10.1007/978-3-030-11072-7_17.

25 Amnon Ta-Shma and Uri Zwick. Deterministic rendezvous, treasure hunts, and strongly universal exploration sequences. *ACM Trans. Algorithms*, 10(3):12:1–12:15, 2014. doi:10.1145/2601068.

Liveness and Latency of Byzantine State-Machine Replication

Manuel Bravo
Informal Systems, Madrid, Spain

Gregory Chockler
University of Surrey, UK

Alexey Gotsman
IMDEA Software Institute, Madrid, Spain

Abstract

Byzantine state-machine replication (SMR) ensures the consistency of replicated state in the presence of malicious replicas and lies at the heart of the modern blockchain technology. Byzantine SMR protocols often guarantee safety under all circumstances and liveness only under synchrony. However, guaranteeing liveness even under this assumption is nontrivial. So far we have lacked systematic ways of incorporating liveness mechanisms into Byzantine SMR protocols, which often led to subtle bugs. To close this gap, we introduce a modular framework to facilitate the design of provably live and efficient Byzantine SMR protocols. Our framework relies on a *view* abstraction generated by a special *SMR synchronizer* primitive to drive the agreement on command ordering. We present a simple formal specification of an SMR synchronizer and its bounded-space implementation under partial synchrony. We also apply our specification to prove liveness and analyze the latency of three Byzantine SMR protocols via a uniform methodology. In particular, one of these results yields what we believe is the first rigorous liveness proof for the algorithmic core of the seminal PBFT protocol.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Replication, blockchain, partial synchrony, liveness

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.12

Related Version *Extended Version*: https://arxiv.org/abs/2202.06679 [17]

Funding This work was partially supported by an ERC Starting Grant RACCOON and by a research grant from Nomadic Labs and the Tezos Foundation.

Acknowledgements We thank the following people for comments that helped improve the paper: Lăcrămioara Aştefănoaei, Hagit Attiya, Alysson Besani, Armando Castañeda, Peter Davies, Dan O'Keeffe, Idit Keidar, Giuliano Losa, Alejandro Naser, and Eugen Zălinescu.

1 Introduction

Byzantine state-machine replication (SMR) [52] ensures the consistency of replicated state even when some of the replicas are malicious. It lies at the heart of the modern blockchain technology and is closely related to the classical Byzantine consensus problem. Unfortunately, no deterministic protocol can guarantee both safety and liveness of Byzantine SMR when the network is asynchronous [33]. A common way to circumvent this while maintaining determinism is to guarantee safety under all circumstances and liveness only under synchrony. This is formalized by the *partial synchrony* model [26, 32], which stipulates that after some unknown *Global Stabilization Time* (GST) the system becomes synchronous, with message delays bounded by an unknown constant δ and process clocks tracking real time. Before GST messages can be lost or delayed, and clocks at different processes can drift apart.

© Manuel Bravo, Gregory Chockler, and Alexey Gotsman;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 12; pp. 12:1–12:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Historically, researchers have paid more attention to safety of Byzantine SMR protocols than their liveness. For example, while the seminal PBFT protocol came with a detailed safety proof [23, §A], the nontrivial mechanisms ensuring its liveness were only given a brief informal justification [25, §4.5.1], which did not cover their most critical properties. However, ensuring liveness under partial synchrony is far from trivial, as illustrated by the many liveness bugs found in existing protocols [2, 4, 8, 12, 22]. In particular, classical failure detectors and leader oracles [26, 34] are of little help: while they have been widely used under benign failures [38, 39, 47], their implementations under Byzantine failures are either impractical [43] or detect only restricted failure types [30, 40, 46]. As an alternative, a textbook by Cachin et al. [20] proposed a leader oracle-like abstraction that accepts hints from the application to identify potentially faulty processes. However, as we explain in §8 and [17, §F], their specification of the abstraction is impossible to implement, and in fact, the consensus algorithm constructed using it in [20] also suffers from a liveness bug.

Recent work on ensuring liveness has departed from failure detectors and instead revisited the approach of the original DLS paper [32]. This exploits the common structure of Byzantine consensus and SMR protocols under partial synchrony: such protocols usually divide their execution into *views*, each with a designated leader process that coordinates the protocol execution. If the leader is faulty, the processes switch to another view with a different leader. To ensure liveness, an SMR protocol needs to spend sufficient time in views that are entered by all correct processes and where the leader correctly follows the protocol. The challenge of achieving such *view synchronization* is that, before GST, clocks can diverge and messages that could be used to synchronize processes can get lost or delayed; even after GST, Byzantine processes may try to disrupt attempts to bring everybody into the same view. *View synchronizers* [16, 48, 49, 57] encapsulate mechanisms for dealing with this challenge, allowing them to be reused across protocols.

View synchronizers have been mostly explored in the context of (single-shot) Byzantine consensus. In this case a synchronizer can just switch processes through an infinite series of views, so that eventually there is a view with a correct leader that is long enough to reach a decision [16, 49]. However, using such a synchronizer for SMR results in suboptimal solutions. For example, one approach is to use the classical SMR construction where each command is decided using a separate black-box consensus instance [52], implemented using a view synchronizer. However, this would force the processes in every instance to iterate over the same sequence of potentially bad views until the one with a correct leader and sufficiently long duration could be reached. As we discuss in §8, other approaches for implementing SMR based on this type of synchronizers also suffer from drawbacks.

To minimize the overheads of view synchronization, instead of automatically switching processes through views based on a fixed schedule, implementations such as PBFT allow processes to stay in the same view for as long as they are happy with its performance. The processes can then reuse a single view to decide multiple commands, usually with the same leader. To be useful for such SMR protocols, a synchronizer needs to allow the processes to control when they want to switch views via a special `advance` call. We call such a primitive an *SMR synchronizer*, to distinguish it from the less flexible *consensus synchronizer* introduced above. This kind of synchronizers was first introduced in [48, 49], but only used as an intermediate module to implement a consensus synchronizer.

In this paper we show that SMR synchronizers can be *directly* exploited to construct efficient and provably live SMR protocols and develop a general blueprint that enables such constructions. In more detail:

- We propose a formal specification of an SMR synchronizer (§3), which is simpler and more general than prior proposals [48, 49]. It is also strictly stronger than the consensus synchronizer of [16], which can be obtained from the SMR synchronizer at no extra cost. Informally, our specification guarantees that *(a)* the system will move to a new view if enough correct processes call `advance`, and *(b)* all correct processes will enter the new view, provided that for long enough, no correct process that enters this view asks to leave it. These properties enable correct processes to iterate through views in search of a well-behaved leader, and to synchronize in a view they are happy with.
- We give an SMR synchronizer implementation and prove that it satisfies our specification (§3.1). Unlike prior implementations [49], ours tolerates message loss before GST while using only bounded space; in practice, this feature is essential to defend against denial-of-service attacks. We also provide a precise latency analysis of our synchronizer, quantifying how quickly all correct processes enter the next view after enough of them call `advance`.
- We demonstrate the usability of our synchronizer specification by applying it to construct and prove the correctness of several SMR protocols. First, we prove the liveness of a variant of PBFT using an SMR synchronizer (§4-5): to the best of our knowledge, this is the first rigorous proof of liveness for PBFT's algorithmic core. The proof establishes a strong liveness guarantee that implies censorship-resistance: every command submitted by a correct process will be executed. The use of the synchronizer specification in the proof allows us to abstract from view synchronization mechanics and focus on protocol-specific reasoning. This reasoning is done using a reusable methodology based on showing that the use of timers in the SMR protocol and the synchronizer together establish properties similar to those of failure detectors. The methodology also handles the realistic ways in which protocols such as PBFT adapt their timeouts to the unknown message delay δ. We demonstrate the generality of our methodology by also applying it to a version of PBFT with periodic leader changes [28, 55, 56] and a HotStuff-like protocol [57] (§7).
- We exploit the latency bounds for our synchronizer to establish both bad-case and good-case bounds for variants of PBFT implemented on top of it (§6). Our bad-case bound assumes that the protocol starts before GST; it shows that after GST all correct processes synchronize in the same view within a bounded time. This time is proportional to a conservatively chosen constant Δ that bounds post-GST message delays in all executions [41, 50]. Our good-case bound quantifies decision latency when the protocol starts after GST and matches the lower bound of [5].

2 System Model

We consider a system of $n = 3f + 1$ processes. At most f of these can be Byzantine (aka *faulty*), i.e., can behave arbitrarily. The rest of the processes are *correct* and we denote their set by $\mathcal{C}$. We call a set Q of $2f + 1$ processes a *quorum* and write $\mathsf{quorum}(Q)$. We assume standard cryptographic primitives [20, §2.3]: processes can communicate via authenticated point-to-point links, sign messages using digital signatures, and use a collision-resistant hash function $\mathsf{hash}()$. We denote by $\langle m \rangle_i$ a message m signed by process p_i.

We consider a *partial synchrony* model [26, 32]: for each execution of the protocol, there exist a time GST and a duration δ such that after GST message delays between correct processes are bounded by δ; before GST messages can get arbitrarily delayed or lost. As in [26], we assume that the values of GST and δ are unknown to the protocol. This reflects the requirements of practical systems, whose designers cannot accurately predict when network problems leading to asynchrony will stop and what the latency will be during the following synchronous period. We also assume that processes have hardware clocks that can drift unboundedly from real time before GST, but do not drift thereafter.

1. **Monotonicity.** A process enters increasing views:
$\forall i, v, v'.\, E_i(v){\downarrow} \wedge E_i(v'){\downarrow} \implies (v < v' \iff E_i(v) < E_i(v'))$
2. **Validity.** A process only enters $v+1$ if some correct process has attempted to advance from v:
$\forall i, v.\, E_i(v+1){\downarrow} \implies A_{\text{first}}(v){\downarrow} \wedge A_{\text{first}}(v) < E_i(v+1)$
3. **Bounded Entry.** For some $\mathcal{V}$ and d, if a process enters a view $v \geq \mathcal{V}$ and no process attempts to advance to a higher view within time d, then all correct processes will enter v within d:
$\exists \mathcal{V}, d. \forall v \geq \mathcal{V}.\, E_{\text{first}}(v){\downarrow} \wedge \neg(A_{\text{first}}(v) < E_{\text{first}}(v) + d) \implies (\forall p_i \in \mathcal{C}.\, E_i(v){\downarrow}) \wedge E_{\text{last}}(v) \leq E_{\text{first}}(v) + d$
4. **Startup.** Some correct process will enter view 1 if $f+1$ processes call `advance`:
$(\exists P \subseteq \mathcal{C}.\, |P| = f+1 \wedge (\forall p_i \in P.\, A_i(0){\downarrow})) \implies E_{\text{first}}(1){\downarrow}$
5. **Progress.** If a correct process enters a view v and, for some set P of $f+1$ correct processes, any process in P that enters v eventually calls `advance`, then some correct process will enter $v+1$:
$\forall v.\, E_{\text{first}}(v){\downarrow} \wedge (\exists P \subseteq \mathcal{C}.\, |P| = f+1 \wedge (\forall p_i \in P.\, E_i(v){\downarrow} \implies A_i(v){\downarrow})) \implies E_{\text{first}}(v+1){\downarrow}$

Figure 1 SMR synchronizer specification.

3 SMR Synchronizer Specification and Implementation

We consider a synchronizer interface defined in [48, 49], which here we call an *SMR synchronizer*. Let $\mathsf{View} = \{1, 2, \dots\}$ be the set of *views*, ranged over by v; we use 0 to denote an invalid initial view. The synchronizer produces notifications `new_view`(v) at a process, telling it to *enter* view v. To trigger these, the synchronizer allows a process to call a function `advance`(), which signals that the process wishes to *advance* to a higher view. We assume that a correct process does not call `advance` twice without an intervening `new_view` notification.

Our first contribution is the SMR synchronizer specification in Figure 1, which is simpler and more general than prior proposals [48, 49] (see §8 for a discussion). The specification relies on the following notation. Given a view v entered by a correct process p_i, we denote by $E_i(v)$ the time when this happens; we let $E_{\text{first}}(v)$ and $E_{\text{last}}(v)$ denote respectively the earliest and the latest time when some correct process enters v. We denote by $A_i(v)$ the time when a correct process p_i calls `advance` while in v, and let $A_{\text{first}}(v)$ and $A_{\text{last}}(v)$ denote respectively the earliest and the latest time when this happens. Given a partial function f, we write $f(x){\downarrow}$ if $f(x)$ is defined, and $f(x){\uparrow}$ if $f(x)$ is undefined.

The Monotonicity property in Figure 1 ensures that views can only increase at a given process. Validity ensures that a process may only enter a view $v+1$ if some correct process has called `advance` in v. This prevents faulty processes from disrupting the system by forcing view changes. As a corollary of Validity we can prove that, if a view v' is entered by some correct process, then so are all the views v preceding v'.

▶ **Proposition 1.** $\forall v, v'.\, 0 < v < v' \wedge E_{\text{first}}(v'){\downarrow} \implies E_{\text{first}}(v){\downarrow} \wedge E_{\text{first}}(v) < E_{\text{first}}(v')$.

Proof. Fix $v' \geq 2$ and assume that a correct process enters v', so that $E_{\text{first}}(v'){\downarrow}$. We prove by induction on k that $\forall k = 0..(v'-1).\, E_{\text{first}}(v'-k){\downarrow} \wedge E_{\text{first}}(v'-k) \leq E_{\text{first}}(v')$. The base case of $k=0$ is trivial. For the inductive step, assume that the required holds for some k. Then by Validity there exists a time $t < E_{\text{first}}(v'-k)$ at which some correct process p_j attempts to advance from $v'-k-1$. But then p_j's view at t is $v'-k-1$. Hence, p_j enters $v'-k-1$ before t, so that $E_{\text{first}}(v'-k-1) < t < E_{\text{first}}(v'-k) \leq E_{\text{first}}(v')$, as required. ◀

Bounded Entry ensures that, if some process enters view v, then all correct processes will do so within at most d time units of each other ($d = 2\delta$ for our implementation). This only holds if within d no process attempts to advance to a higher view, as this may make some processes skip v and enter a higher view directly. Bounded Entry also holds only starting from some view $\mathcal{V}$, since we may not be able to guarantee it for views entered before GST.

```
1 when the process starts or timer expires
2   advance();
3 upon new_view(v)
4   stop_timer(timer);
5   start_timer(timer, τ);
```

Figure 2 A simple client of the SMR synchronizer.

Startup ensures that some correct process enters view 1 if $f+1$ processes call `advance`. Given a view v entered by a correct process, Progress determines conditions under which some correct process will enter the next view $v+1$. This will happen if for some set P of $f+1$ correct processes, any process in P entering v eventually calls `advance`. Note that even a single `advance` call at a correct process *may* lead to a view switch (reflecting the fact that in implementations faulty processes may help this correct process). Startup and Progress ensure that the synchronizer *must* switch if at least $f+1$ correct processes ask for this. We now illustrate a typical pattern of their use, which we later apply to PBFT (§5). To this end, we consider a simple client in Figure 2, where in each view a process sets a timer for a fixed duration τ and calls `advance` when the timer expires. Using Startup and Progress we prove that this client keeps switching views forever as follows.

▶ **Proposition 2.** *In any execution of the client in Figure 2:* $\forall v.\, \exists v'.\, v' > v \wedge E_{\text{first}}(v')\downarrow$.

Proof. Since all correct processes initially call `advance`, by Startup some correct process eventually enters view 1. Assume now that the proposition is false, so that there is a maximal view v entered by any correct process. Let P be any set of $f+1$ correct processes and consider an arbitrary process $p_i \in P$ that enters v. When this happens, p_i sets the timer for the duration τ. The process then either calls `advance` when timer expires, or enters a new view v' before this. In the latter case $v' > v$ by Monotonicity, which is impossible. Hence, p_i eventually calls `advance` while in v. Since p_i was chosen arbitrarily, $\forall p_i \in P.\, E_i(v)\downarrow \implies A_i(v)\downarrow$. Then by Progress we get $E_{\text{first}}(v+1)\downarrow$: a contradiction. ◀

Similarly to Figure 2, we can use an SMR synchronizer satisfying the properties in Figure 1 to implement a *consensus synchronizer* [16, 49] without extra overhead. This lacks an `advance` call and provides only the `new_view` notification, which it keeps invoking at increasing intervals so that eventually the there is a view long enough for the consensus protocol running on top to decide. We obtain a consensus synchronizer if in Figure 2 we propagate the `new_view` notification to the consensus protocol and set the timer to an unboundedly increasing function of views instead of a constant τ. In [17, §A] we show that the resulting consensus synchronizer satisfies the specification proposed in [16].

3.1 A Bounded-Space SMR Synchronizer

We now present a bounded-space algorithm that implements the specification in Figure 1 under partial synchrony for $d = 2\delta$. Our implementation reuses algorithmic techniques from a consensus synchronizer of Bravo et al. [16]. However, it supports a more general abstraction, and thus requires a more intricate correctness proof and latency analysis (§3.2).

When a process calls `advance` (line 1), the synchronizer does not immediately move to the next view v', but disseminates a `WISH`(v') message announcing its intention. A process enters a new view once it accumulates a sufficient number of `WISH` messages supporting this. A naive synchronizer design could follow Bracha broadcast [15]: enter a view v' upon receiving

```
1  function advance()
2      send WISH(max(view + 1, view⁺))
         to all;
3      advanced ← TRUE;

4  periodically                ▷ every ρ time units
5      if advanced then
6          send WISH(max(view + 1, view⁺))
             to all;
7      else if view⁺ > 0 then
8          send WISH(view⁺) to all;

9  when received WISH(v) from p_j
10     prev_v, prev_v⁺ ← view, view⁺;
11     if v > max_views[j] then max_views[j] ← v;
12     view  ← max{v | ∃k. max_views[k] = v ∧
                   |{j | max_views[j] ≥ v}| ≥ 2f + 1};
13     view⁺ ← max{v | ∃k. max_views[k] = v ∧
                   |{j | max_views[j] ≥ v}| ≥ f + 1};
14     if view⁺ = view ∧ view > prev_v then
15         trigger new_view(view);
16         advanced ← FALSE;
17     if view⁺ > prev_v⁺ then
18         send WISH(view⁺) to all
```

Figure 3 A bounded-space SMR synchronizer. All counters are initially 0.

$2f+1$ WISH(v') messages, and echo WISH(v') upon receiving $f+1$ copies thereof; the latter is needed to combat equivocation by Byzantine processes. However, in this case the process would have to track all newly proposed views for which $< 2f+1$ WISHes have been received. Since messages sent before GST can be lost or delayed, this would require unbounded space. To reduce the space complexity, in our algorithm a process only remembers the highest view received from each process, kept in an array max_views (line 11). Variables view and view^+ respectively hold the $(2f+1)$st highest and the $(f+1)$st highest views in max_views (lines 12-13). These variables never decrease and always satisfy $\text{view} \le \text{view}^+$.

The process enters the view stored in view when this variable increases (line 15). A process thus enters v only if it receives $2f+1$ WISHes for views $\ge v$, and a process may be forced to switch views even if it did not call `advance`; the latter helps lagging processes to catch up. The variable view^+ increases when the process receives $f+1$ WISHes for views $\ge \text{view}^+$, and thus some correct process wishes to enter a new view $\ge \text{view}^+$. In this case we echo view^+ (line 18), to help other processes switch views and satisfy Bounded Entry and Progress.

The guard $\text{view}^+ = \text{view}$ in line 14 ensures that a process does not enter a "stale" view such that another correct process already wishes to enter a higher one. Similarly, when the process calls `advance`, it sends a WISH for the maximum of $\text{view}+1$ and view^+ (line 2). Thus, if $\text{view} = \text{view}^+$, so that the values of the two variables have not changed since the process entered the current view, then the process sends a WISH for the next view ($\text{view}+1$). Otherwise, $\text{view} < \text{view}^+$, and the process sends a WISH for the higher view view^+. Finally, to deal with message loss before GST, a process retransmits the highest WISH it sent every ρ time units, according to its local clock (line 4). Depending on whether the process has called `advance` in the current view (tracked by advanced), the WISH is computed as in lines 18 or 2.

Our SMR synchronizer requires only $O(n)$ variables for storing views. Proposition 1 also ensures that views entered by correct processes do not skip values, which limits the power of the adversary to exhaust their allocated space (similarly to [11]).

3.2 SMR Synchronizer Correctness and Latency Bounds

The following theorem (proved in [17, §B]) states the correctness of our synchronizer as well as and its performance properties. In §6 we apply the latter to bound the latency of Byzantine SMR protocols. Given a view v that was entered by a correct process p_i, we let $T_i(v)$ denote the time at which p_i either attempts to advance from v or enters a view $> v$; we let $T_{\text{last}}(v)$ denote the latest time when a correct process does so. We assume that every correct process eventually attempts to advance from view 0 unless it enters a view > 0, i.e., $\forall p_i \in \mathcal{C}.\, T_i(0){\downarrow}$.

▶ **Theorem 3.** *Consider an execution with an eventual message delay δ. The algorithm in Figure 3 satisfies the properties in Figure 1 for $d = 2\delta$ and $\mathcal{V} = \max\{v \mid (E_{\text{first}}(v){\downarrow} \wedge E_{\text{first}}(v) < \mathsf{GST} + \rho) \vee v = 0\} + 1$ if $A_{\text{first}}(0) < \mathsf{GST}$, and $\mathcal{V} = 1$, otherwise. Furthermore:*

A. $\forall v.\, E_{\text{first}}(v){\downarrow} \wedge A_{\text{first}}(0) < \mathsf{GST} \implies E_{\text{last}}(v) \le \max(E_{\text{first}}(v), \mathsf{GST} + \rho) + 2\delta.$

B. $\forall v.\, E_{\text{first}}(v+1){\downarrow} \implies E_{\text{last}}(v+1) \le \begin{cases} \max(T_{\text{last}}(v), \mathsf{GST} + \rho) + \delta, & \text{if } A_{\text{first}}(0) < \mathsf{GST}; \\ T_{\text{last}}(v) + \delta, & \text{otherwise.} \end{cases}$

The theorem gives a witness for $\mathcal{V}$ in Bounded Entry: it is the next view after the highest one entered by a correct process at or before $\mathsf{GST}+\rho$ (or 1 if no view was entered). Property A bounds the latest time any correct process can enter a view that has been previously entered by a correct process. It is similar to Bounded Entry, but also handles views $< \mathcal{V}$. Property B refines Progress: while the latter guarantees that the synchronizer will enter $v + 1$ if enough processes ask for this, the former bounds the time by which this will happen.

4 PBFT Using an SMR Synchronizer

We now demonstrate how an SMR synchronizer can be used to implement Byzantine SMR. More formally, we implement *Byzantine atomic broadcast* [20], from which SMR can be implemented in the standard way [52]. This allows processes to broadcast *values*, and we assume an application-specific predicate to indicate whether a value is valid [21] (e.g., a block in a blockchain is invalid if it lacks correct signatures). We assume that all values broadcast by correct processes in a single execution are valid and unique. Then Byzantine atomic broadcast is defined by the following properties:

- **Integrity.** Every process delivers a value at most once.
- **External Validity.** A correct process delivers only values satisfying valid().
- **Ordering.** If a correct process p delivers x_1 before x_2, then another correct process q cannot deliver x_2 before x_1.
- **Liveness.** If a correct process broadcasts or delivers x, then eventually all correct processes will deliver x. (Note that this implies *censorship-resistance*: the service cannot selectively omit values submitted by correct processes.)

The PBFT-light protocol. We implement Byzantine atomic broadcast in a *PBFT-light* protocol (Figures 4-6), which faithfully captures the algorithmic core of the seminal Practical Byzantine Fault Tolerance protocol (PBFT) [24]. Whereas PBFT integrated view synchronization functionality with the core SMR protocol, PBFT-light delegates this to an SMR synchronizer, and in §5 we rigorously prove its liveness when using any synchronizer satisfying our specification. When using the synchronizer in Figure 3, the protocol also incurs only bounded space overhead (see [17, §C.4] for details).

We base PBFT-light on the PBFT protocol with signatures and, for simplicity, omit the mechanisms for managing checkpoints and watermarks; these can be easily added without affecting liveness. The protocol works in a succession of views produced by the synchronizer. A process stores its current view in curr_view. Each view v has a fixed *leader* $\mathsf{leader}(v) = p_{((v-1) \bmod n)+1}$ that is responsible for totally ordering values submitted for broadcast; the other processes are *followers*, which vote on proposals made by the leader. Processes store the sequence of (unique) values proposed by the leader in a log array; at the leader, a next counter points to the first free slot in the array. Processes monitor the leader's behavior and ask the synchronizer to advance to another view if they suspect that the leader is faulty. A status variable records whether the process is operating as normal in the current view (NORMAL) or is changing the view.

1 **function** start()
2   **if** curr_view $= 0$ **then** advance();

3 **when a timer expires**
4   stop_all_timers();
5   advance();
6   status $\leftarrow$ ADVANCED;
7   dur_delivery $\leftarrow$ dur_delivery $+ \tau$;
8   dur_recovery $\leftarrow$ dur_recovery $+ \tau$;

9 **function** broadcast(x)
10   **pre:** valid(x);
11   **send** $\langle$BROADCAST$(x)\rangle_i$ **to all periodically until** x **is delivered**

12 **when received** BROADCAST(x)
13   **pre:** valid(x) $\wedge$ status $=$ NORMAL $\wedge$ (timer_delivery[x] not active) $\wedge$ ($\forall k.\, k \leq$ last_delivered $\Longrightarrow$ commit_log[k] $\neq x$);
14   start_timer(timer_delivery[x], dur_delivery);
15   **send** $\langle$FORWARD$(x)\rangle_i$ **to** leader(curr_view);

16 **when received** FORWARD(x)
17   **pre:** valid(x) $\wedge$ status $=$ NORMAL $\wedge$ $p_i =$ leader(curr_view) $\wedge$ $\forall k.$ log[k] $\neq x$;
18   **send** $\langle$PREPREPARE(curr_view, next, x)$\rangle_i$ **to all**;
19   next $\leftarrow$ next $+ 1$;

20 **when received** $\langle$PREPREPARE$(v, k, x)\rangle_j$
21   **pre:** $p_j =$ leader(v) $\wedge$ curr_view $= v$ $\wedge$ status $=$ NORMAL $\wedge$ phase[k] $=$ START $\wedge$ valid(x) $\wedge$ ($\forall k'.$ log[k'] $\neq x$)
22   (log, phase)[k] $\leftarrow$ (x, PREPREPARED);
23   **send** $\langle$PREPARE$(v, k,$ hash$(x))\rangle_i$ **to all**;

24 **when received** $\{\langle$PREPARE$(v, k, h)\rangle_j \mid p_j \in Q\} = C$ **for a quorum** Q
25   **pre:** curr_view $= v$ $\wedge$ phase[k] $=$ PREPREPARED $\wedge$ status $=$ NORMAL $\wedge$ hash(log[k]) $= h$;
26   (prep_log, prep_view, cert, phase)[k] $\leftarrow$ (log[k], curr_view, C, PREPARED);
27   **send** $\langle$COMMIT$(v, k, h)\rangle_i$ **to all**;

28 **when received** $\{\langle$COMMIT$(v, k, h)\rangle_j \mid p_j \in Q\} = C$ **for a quorum** Q
29   **pre:** curr_view $= v$ $\wedge$ phase[k] $=$ PREPARED $\wedge$ status $=$ NORMAL $\wedge$ hash(prep_log[k]) $= h$;
30   (commit_log, phase)[k] $\leftarrow$ (log[k], COMMITTED);
31   **broadcast** $\langle$DECISION(commit_log[k], k, C);

32 **when** commit_log[last_delivered $+ 1$] $\neq \bot$
33   last_delivered $\leftarrow$ last_delivered $+ 1$;
34   **if** commit_log[last_delivered] $\neq$ nop **then**
35   deliver(commit_log[last_delivered])
36   stop_timer(timer_delivery[commit_log[last_delivered]]);
37   **if** last_delivered $=$ init_log_length $\wedge$ status $=$ NORMAL **then**
38   stop_timer(timer_recovery);

39 **when received** DECISION(x, k, C)
40   **pre:** commit_log[k] $\neq \bot$ $\wedge$ $\exists v.$ committed(C, v, k, hash(x));
41   commit_log[k] $\leftarrow x$;

Figure 4 Normal operation of PBFT-light at a process p_i.

Normal protocol operation. A process broadcasts a valid value x using a broadcast function (line 9). This keeps sending the value to all processes in a BROADCAST message until the process delivers the value, to tolerate message loss before GST. When a process receives a BROADCAST message with a new value (line 12), it forwards the value to the leader in a FORWARD message. This ensures that the value reaches the leader even when broadcast by a faulty process, which may withhold the BROADCAST message from the leader. (We explain the timer set in line 14 later.) When the leader receives a new value x in a FORWARD message (line 16), it sends a PREPREPARE message to all processes (including itself) that includes x and its position in the log, generated from the next counter. Processes vote on the leader's proposal in two phases. Each process keeps track of the status of values going through the vote in an array phase, whose entries initially store START.

When a process receives a proposal x for a position k from the leader of its view v (line 20), it first checks that phase[k] $=$ START, so that it has not yet accepted a proposal for the position k in the current view. It also checks that the value is valid and distinct from all values it knows about. The process then stores x in log[k] and advances phase[k] to PREPREPARED. Since a faulty leader may send different proposals for the same position to different processes, the process next communicates with others to check that they received the same proposal.

To this end, it disseminates a PREPARE message with the position and the hash of the value x it received. The process handles x further once it gathers a set C of PREPARE messages from a quorum matching the value (line 24), which we call a *prepared certificate* and check using the prepared predicate in Figure 6. In this case the process stores the value in prep_log[k], the certificate in cert[k], and the view in which it was formed in prep_view[k]. At this point we say that the process *prepared* the proposal, as recorded by setting its phase to PREPARED. It is easy to show that processes cannot prepare different values at the same position and view, since each correct process can send only one corresponding PREPARE message.

Having prepared a value, the process disseminates a COMMIT message with its hash. Once the process gathers a quorum of matching COMMIT messages (line 28), it stores the value in a commit_log array and advances its phase to COMMITTED: the value is now *committed*. The protocol ensures that correct processes cannot commit different values at the same position, even in different views. We call a quorum of matching COMMIT messages a *commit certificate* and check it using the committed predicate in Figure 6. A process delivers committed values in the commit_log order, with last_delivered tracking the position last delivered position.

To satisfy the Liveness property of atomic broadcast, similarly to [12], PBFT-light allows a process to find out about committed values from other processes directly. When a process commits a value (line 28), it disseminates a DECISION message with the value, its position k in the log and the commit certificate (line 31). A process receiving a DECISION with a valid certificate saves the value in commit_log[k], which allows it to be delivered (line 32). The DECISION messages are disseminated via reliable broadcast ensuring that, if one correct process delivers the value, then so do all others. To implement this, each process could periodically resend the DECISION messages it has (omitted from the pseudocode). A more practical implementation would only resend information that other processes are missing. As proved in [32], such periodic resends are unavoidable in the presence of message loss.

View initialization. When the synchronizer tells a process to move to a new view v (line 42), the process sets curr_view to v, which ensures that it will no longer accept messages from prior views. It also sets status to INITIALIZING, which means that the process is not yet ready to order values in the new view. It then sends a NEW_LEADER message to the leader of v with the information about the values it has prepared so far and their certificates[1].

The new leader waits until it receives a quorum of well-formed NEW_LEADER messages, as checked by the predicate ValidNewLeader (line 48). Based on these, the leader computes the initial log of the new view, stored in log'. Similarly to Paxos [45], for each index k the leader puts at the kth position in log' the value prepared in the highest view (line 50). The resulting array may contain empty or duplicate entries. To resolve this, the leader writes nop into empty entries and those entries for which there is a duplicate prepared in a higher view (line 53). The latter is safe because one can show that no value could have been committed in such entries in prior views. Finally, the leader sends a NEW_STATE message to all processes, containing the initial log and the NEW_LEADER messages from which it was computed (line 56).

A process receiving a NEW_STATE first checks its correctness by redoing the leader's computation (ValidNewState, line 57). If the check passes, the process overwrites its log with the new one and sets status to NORMAL. It also sends PREPARE messages for all log entries, to commit them in the new view. A more practical implementation would include a checkpointing mechanism, so that a process restarts committing previous log entries only from the last stable checkpoint [24]; this mechanism can be easily added to PBFT-light.

[1] In PBFT this information is sent in VIEW-CHANGE messages, which also play a role similar to WISH messages in our synchronizer (Figure 3). In PBFT-light we opted to eschew VIEW-CHANGE messages to maintain a clear separation between view synchronization internals and the SMR protocol.

```
42 upon new_view(v)
43     stop_all_timers();
44     curr_view ← v;
45     status ← INITIALIZING;
46     send ⟨NEW_LEADER(curr_view, prep_view,
         prep_log, cert)⟩_i to leader(curr_view);
47     start_timer(timer_recovery, dur_recovery);

48 when received {⟨NEW_LEADER(v, prep_view_j,
     prep_log_j, cert_j)⟩_j | p_j ∈ Q} = M
     for a quorum Q
49     pre: p_i = leader(v) ∧ curr_view = v ∧
            status = INITIALIZING ∧
            ∀m ∈ M. ValidNewLeader(m);
50     forall k do
51         if ∃p_j' ∈ Q. prep_view_j'[k] ≠ 0 ∧
             ∀p_j ∈ Q. prep_view_j[k] ≤ prep_view_j'[k]
             then log'[k] ← prep_log_j'[k];
52     next ← max{k | log'[k] ≠ ⊥};
53     forall k = 1..(next − 1) do
54         if log'[k] = ⊥ ∨ ∃k'. k' ≠ k ∧
             log'[k'] = log'[k] ∧ ∃p_j' ∈ Q. ∀p_j ∈ Q.
             prep_view_j'[k'] > prep_view_j[k] then
55             log'[k] ← nop
56     send ⟨NEW_STATE(v, log', M)⟩_i to all;

57 when received ⟨NEW_STATE(v, log', M)⟩_j = m
58     pre: status = INITIALIZING ∧
            curr_view = v ∧ ValidNewState(m);
59     log ← log';
60     forall {k | log[k] ≠ ⊥} do
61         phase[k] ← PREPREPARED;
62         send ⟨PREPARE(v, k, hash(log[k]))⟩_i
             to all;
63     status ← NORMAL;
64     init_log_length ← max{k | log[k] ≠ ⊥};
65     if init_log_length ≤ last_delivered then
66         stop_timer(timer_recovery);
```

Figure 5 View-initialization protocol of PBFT-light at a process p_i.

$$\mathsf{prepared}(C, v, k, h) \iff \exists Q.\, \mathsf{quorum}(Q) \land C = \{\langle \mathtt{PREPARE}(v, k, h)\rangle_j \mid p_j \in Q\}$$

$$\mathsf{committed}(C, v, k, h) \iff \exists Q.\, \mathsf{quorum}(Q) \land C = \{\langle \mathtt{COMMIT}(v, k, h)\rangle_j \mid p_j \in Q\}$$

$$\mathsf{ValidNewLeader}(\langle \mathtt{NEW_LEADER}(v, \mathit{prep_view}, \mathit{prep_log}, \mathit{cert})\rangle_{_}) \iff \forall k.\, (\mathit{prep_view}[k] > 0 \implies \mathit{prep_view}[k] < v \land \mathsf{prepared}(\mathit{cert}[k], \mathit{prep_view}[k], k, \mathit{prep_log}[k]))$$

$$\mathsf{ValidNewState}(\langle \mathtt{NEW_STATE}(v, \mathit{log}', M)\rangle_i) \iff p_i = \mathsf{leader}(v) \land \exists Q, \mathit{prep_view}, \mathit{prep_log}, \mathit{cert}.\ \mathsf{quorum}(Q) \land M = \{\langle \mathtt{NEW_LEADER}(v, \mathit{prep_view}_j, \mathit{log}_j, \mathit{cert}_j)\rangle_j \mid p_j \in Q\} \land (\forall m \in C.\, \mathsf{ValidNewLeader}(m)) \land (\mathit{log}' \text{ is computed from } M \text{ as per lines 50-55})$$

Figure 6 Auxiliary predicates for PBFT-light.

Triggering view changes. We now describe when a process calls `advance`, which is key to ensure liveness (§5). This happens either on start-up (line 2) or when the process suspects that the current leader is faulty. To this end, the process monitors the leader's behavior using timers; if one of these expires, the process calls `advance` and sets status to ADVANCED (line 3). First, the process checks that each value it receives is delivered promptly: e.g., to guard against a faulty leader censoring certain values. For a value x this is done using timer_delivery$[x]$, set for a duration dur_delivery when the process receives BROADCAST(x) (lines 14). The timer is stopped when the process delivers x (line 36). A process also checks that the leader initializes a view quickly enough: e.g., to guard against the leader crashing during the initialization. Thus, when a process enters a view it starts timer_recovery for a duration dur_recovery (line 47). The process stops the timer when it delivers all values in the initial log (lines 38 and 66). The above checks may make a process suspect a correct leader if the timeouts are initially set too small with respect to the message delay δ, unknown to the process. To deal with this, a process increases dur_delivery and dur_recovery each time a timer expires, which signals that the current view is not operating normally (lines 7-8).

5 Proving the Liveness of PBFT

Assume that PBFT-light is used with a synchronizer satisfying the specification in Figure 1; to simplify the following latency analysis we let $d = 2\delta$, as for the synchronizer in Figure 3. We now prove that the protocol satisfies the Liveness property of Byzantine atomic broadcast; we defer the proof of the other properties to [17, §C.1]. To the best of our knowledge, this is the first rigorous proof of liveness for the algorithmic core of PBFT: as we elaborate in §8, the liveness mechanisms of PBFT came only with a brief informal justification, which did not cover their most critical properties [25, §4.5.1]. Our proof is simplified by the use of the synchronizer specification, which allows us to abstract from view synchronization mechanics.

We prove the liveness of PBFT-light by showing that the protocol establishes properties reminiscent of those of failure detectors [26]. First, similarly to their completeness property, we prove that every correct process eventually attempts to advance from a *bad* view in which no progress is possible (e.g., because the leader is faulty).

▶ **Lemma 4.** *Assume that a correct process p_i receives* `BROADCAST`(x) *for a valid value x while in a view v. If p_i never delivers x and never enters a view higher than v, then it eventually calls* `advance` *in v.*

The lemma holds because in PBFT-light each process monitors the leader's behavior using timers, and we defer its easy proof to [17, §C.2]. Our next lemma is similar to the eventual accuracy property of failure detectors. It stipulates that if the timeout values are high enough, then eventually any correct process that enters a *good* view (with a correct leader) will never attempt to advance from it. Let $\mathsf{dur_recovery}_i(v)$ and $\mathsf{dur_delivery}_i(v)$ denote respectively the value of dur_recovery and dur_delivery at a correct process p_i while in view v.

▶ **Lemma 5.** *Consider a view $v \geq \mathcal{V}$ such that $E_{\text{first}}(v) \geq \mathsf{GST}$ and $\mathsf{leader}(v)$ is correct. If $\mathsf{dur_recovery}_i(v) > 6\delta$ and $\mathsf{dur_delivery}_i(v) > 4\delta$ at each correct process p_i that enters v, then no correct process calls* `advance` *in v.*

Before proving the lemma, we informally explain the rationale for the bounds on timeouts in it, using the example of dur_recovery. The timer timer_recovery is started at a process p_i when this process enters a view v (line 47), and is stopped when the process delivers all values inherited from previous views (lines 38 or 66). The two events are separated by 4 communication steps of PBFT-light, exchanging messages of the types `NEW_LEADER`, `NEW_STATE`, `PREPARE` and `COMMIT` (Figure 7). However, 4δ would be too small a value for dur_recovery. This is because the leader of v sends its `NEW_STATE` message only after receiving a quorum of `NEW_LEADER` messages, and different processes may enter v and send their `NEW_LEADER` messages at different times (e.g., p_i and p_j in Figure 7). Hence, dur_recovery must additionally accommodate the maximum discrepancy in the entry times, which is $d = 2\delta$ by the Bounded Entry property. Then to ensure that p_i stops the timer before it expires, we require $\mathsf{dur_recovery}_i(v) > 6\delta$. As the above reasoning illustrates, Lemma 5 is more subtle than Lemma 4: while the latter is ensured just by the checks in the SMR protocol, the former relies on the Bounded Entry property of the synchronizer.

Another subtlety about Lemma 5 is that the δ used in its premise is a priori unknown. Hence, to apply the lemma in the liveness proof of PBFT-light, we have to argue that, if correct processes keep changing views due to lack of progress, then all of them will eventually increase their timeouts high enough to satisfy the bounds in Lemma 5. This is nontrivial due to the fact that, as in the original PBFT [23, §2.3.5], in our protocol the processes update their timeouts independently, and may thus disagree on their durations. For example, the

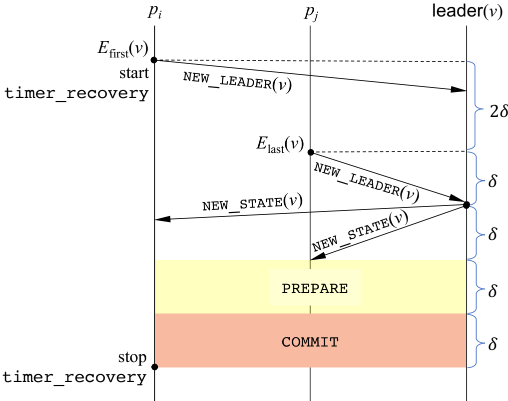


Figure 7 An illustration of the proof of Lemma 5.

first correct process p_i to detect a problem with the current view v will increase its timeouts and call `advance` (line 3). The synchronizer may then trigger `new_view` notifications at other correct processes before they detect the problem as well, so that their timeouts will stay unchanged (line 42). One may think that this allows executions in which only some correct processes keep increasing their timeouts until they are high enough, whereas others are forever stuck with timeouts that are too low, invalidating the premise of Lemma 5. The following lemma rules out such scenarios and also trivially implies Lemma 5. It establishes that, in a sufficiently high view v with a correct leader, if the timeouts at a correct process p_i that enters v are high enough, then this process cannot be the first one to initiate a view change. Hence, for the protocol to enter another view, some other process with lower timeouts must call `advance` and thus increase their durations (line 3).

▶ **Lemma 6.** *Let $v \geq \mathcal{V}$ be such that $E_{\text{first}}(v) \geq \mathsf{GST}$ and $\mathsf{leader}(v)$ is correct, and consider a correct process p_i that enters v. If $\mathsf{dur_recovery}_i(v) > 6\delta$ and $\mathsf{dur_delivery}_i(v) > 4\delta$ then p_i is not the first correct process to call* `advance` *in v.*

Proof. Since $E_{\text{first}}(v) \geq \mathsf{GST}$, messages sent by correct processes after $E_{\text{first}}(v)$ get delivered to all correct processes within δ and process clocks track real time. By contradiction, assume that p_i is the first correct process to call `advance` in v. This happens because a timer expires at p_i. Here we only consider the case when it is timer_recovery, and handle timer_delivery in [17, §C.2]. A process starts timer_recovery when it enters the view v (line 47), and hence, at $E_{\text{first}}(v)$ at the earliest (Figure 7). Because p_i is the first correct process to call `advance` in v and $\mathsf{dur_recovery}_i(v) > 6\delta$, no correct process calls `advance` in v until after $E_{\text{first}}(v) + 6\delta$. Then by Bounded Entry all correct processes enter v by $E_{\text{first}}(v) + 2\delta$. Also, by Validity no correct process can enter $v + 1$ until after $E_{\text{first}}(v) + 6\delta$, and by Proposition 1 the same holds for any view $> v$. Thus, all correct processes stay in v at least until $E_{\text{first}}(v) + 6\delta$.

When a correct process enters v, it sends a `NEW_LEADER` message to the leader of v, which happens by $E_{\text{first}}(v)+2\delta$. When the leader receives such messages from a quorum of processes, it broadcasts a `NEW_STATE` message. Thus, by $E_{\text{first}}(v) + 4\delta$ all correct processes receive this message and set $\mathsf{status} = \text{NORMAL}$. If at that point $\mathsf{init_log_length} \leq \mathsf{last_delivered}$ at p_i, then the process stops timer_recovery (line 66), which contradicts our assumption. Hence, $\mathsf{init_log_length} > \mathsf{last_delivered}$. When a correct process receives `NEW_STATE`, it sends `PREPARE` messages for all positions $\leq \mathsf{init_log_length}$ (line 62). It then takes the correct processes at most 2δ to exchange the sequence of `PREPARE` and `COMMIT` messages that commits the values at all positions $\leq \mathsf{init_log_length}$. Thus, by $E_{\text{first}}(v) + 6\delta$ the process p_i commits and delivers all these positions, stopping timer_recovery (line 38): a contradiction. ◀

▶ **Theorem 7.** *PBFT-light satisfies the Liveness property of Byzantine atomic broadcast.*

Proof. Consider a valid value x broadcast by a correct process. We first prove that x is eventually delivered by some correct process. Assume the contrary. We show:

▷ Claim 1. Every view is entered by some correct process.

Proof. Since all correct processes call `start` (line 1), by Startup a correct process eventually enters some view. We now show that correct processes keep entering new views forever (analogously to the proof of Proposition 2 in §3). Assume that this is false, so that there exists a maximal view v entered by any correct process. Let P be any set of $f+1$ correct processes and consider an arbitrary process $p_i \in P$ that enters v. The process that broadcast x is correct, and thus keeps broadcasting x until the value is delivered (line 11). Since x is never delivered, p_i is guaranteed to receive x while in v. Then by Lemma 4, p_i eventually calls `advance` while in v. Since p_i was picked arbitrarily, we have $\forall p_i \in P.\, E_i(v)\downarrow \implies A_i(v)\downarrow$. Then by Progress we get $E_{\text{first}}(v+1)\downarrow$, which yields a contradiction. Thus, correct processes keep entering views forever. The claim then follows from Proposition 1. ◁

Let view v_1 be the first view such that $v_1 \geq \mathcal{V}$ and $E_{\text{first}}(v_1) \geq \mathsf{GST}$; such a view exists by Claim 1. The next claim is needed to show that all correct processes will increase their timeouts high enough to satisfy the bounds in Lemma 5.

▷ Claim 2. Every correct process calls the timer expiration handler (line 3) infinitely often.

Proof. Assume the contrary and let C_{fin} and C_{inf} be the sets of correct processes that call the timer expiration handler finitely and infinitely often, respectively. Then $C_{\text{fin}} \neq \emptyset$, and by Claim 1 and Validity, $C_{\text{inf}} \neq \emptyset$. The values of dur_delivery and dur_recovery increase unboundedly at processes from C_{inf}, and do not change after some view v_2 at processes from C_{fin}. By Claim 1 and since leaders rotate round-robin, there is a view $v_3 \geq \max\{v_2, v_1\}$ with a correct leader such that any process $p_i \in C_{\text{inf}}$ that enters v_3 has $\mathsf{dur_delivery}_i(v_3) > 4\delta$ and $\mathsf{dur_recovery}_i(v_3) > 6\delta$. By Claim 1 and Validity, at least one correct process calls `advance` in v_3; let p_l be the first process to do so. Since $v_3 \geq v_2$, p_l cannot be in C_{fin} because none of these processes increase their timers in v_3. Then $p_l \in C_{\text{inf}}$, contradicting Lemma 6. ◁

By Claims 1 and 2, there exists a view $v_4 \geq v_1$ with a correct leader such that some correct process enters v_4, and for any correct process p_i that enters v_4 we have $\mathsf{dur_delivery}_i(v_4) > 4\delta$ and $\mathsf{dur_recovery}_i(v_4) > 6\delta$. By Lemma 5, no correct process calls `advance` in v_4. Then, by Validity, no correct process enters $v_4 + 1$, which contradicts Claim 1. This contradiction shows that x must be delivered by a correct process. Then, since the protocol reliably broadcasts committed values (line 31), all correct processes will also eventually deliver x. ◀

6 Latency Bounds for PBFT

Assume that PBFT-light is used with our SMR synchronizer in Figure 3. We now quantify its latency using the bounds for the synchronizer in Theorem 3, yielding the first detailed latency analysis for a PBFT-like protocol. Due to space constraints we defer proofs to [17, §C.3]. To state our bounds, we assume the existence of a known upper bound Δ on the maximum value of δ in any execution [41, 50], so that we always have $\delta < \Delta$. In practice, Δ provides a conservative estimate of the message delay during synchronous periods, which may be much higher than the maximal delay δ in a particular execution. We modify the protocol in Figure 4 so that in lines 7-8 it does not increase dur_recovery and dur_delivery above 6Δ and 4Δ, respectively. This corresponds to the bounds in Lemma 5 and preserves the protocol

liveness. Finally, we assume that periodic handlers (line 4 in Figure 3 and line 11 in Figure 4) are executed every ρ time units, and that the latency of reliable broadcast in line 31 under synchrony is $\leq \delta+\rho$ (this corresponds to an implementation that just periodically retransmits `DECISION` messages).

We quantify the latency of PBFT-light in both bad and good cases. For the bad case we assume that the protocol starts during the asynchronous period. Given a value x broadcast before GST, we quantify how quickly after GST all correct processes deliver x. For simplicity, we assume that timeouts are high enough at GST and that leader($\mathcal{V}$) is correct.

▶ **Theorem 8.** *Assume that before* GST *all correct processes start executing the protocol and one of them broadcasts x. Let $\mathcal{V}$ be defined as in Theorem 3 and assume that* leader($\mathcal{V}$) *is correct and at* GST *each correct process has* dur_recovery $> 6\delta$ *and* dur_delivery $> 4\delta$*. Then all correct processes deliver x by* $\mathsf{GST} + \rho + \max\{\rho+\delta, 6\Delta\} + 4\Delta + \max\{\rho, \delta\} + 7\delta$.

Although the latency bound looks complex, its main message is simple: PBFT-light recovers after a period of asynchrony in bounded time. This time is dominated by multiples of Δ; without the assumption that leader($\mathcal{V}$) is correct it would also be multiplied by f due to going over up to f views with faulty leaders. In [17, §C.3] we show the bound using the latency guarantees of our synchronizer (Properties A and B in Theorem 3).

We now consider the case when the protocol starts during the synchronous period, i.e., after GST. The following theorem quantifies how quickly all correct processes enter the first functional view, which in this case is view 1. If leader(1) is correct, it also quantifies how quickly a broadcast value x is delivered by all correct processes. The bound takes into account the following optimization: in view 1 the processes do not need to exchange `NEW_LEADER` messages. Then, after the systems starts up, the protocol delivers values within 4δ, which matches an existing lower bound of 3δ for the delivery time starting from the leader [5].

▶ **Theorem 9.** *Assume that all correct processes start the protocol after* GST *with* dur_recovery $> 5\delta$ *and* dur_delivery $> 4\delta$*. Then the $\mathcal{V}$ defined in Theorem 3 is equal to 1 and $E_{\text{last}}(1) \leq T_{\text{last}}(0) + \delta$. Furthermore, if a correct process broadcasts x at $t \geq$* GST *and* leader(1) *is correct, then all correct processes deliver x by* $\max\{t, T_{\text{last}}(0) + \delta\} + 4\delta$.

7 Additional Case Studies

To demonstrate the generality of SMR synchronizers, we have also used it to ensure the liveness of two other protocols. First, we handle a variant of PBFT that periodically forces a leader change, as is common in modern Byzantine SMR [28, 55, 56]. In this protocol a process calls `advance` not only when it suspects the current leader to be faulty, but also when it delivers B values proposed by this leader (for a fixed B). Second, we have applied the SMR synchronizer to a variant of the above protocol that follows the approach of HotStuff [57]. The resulting protocol adds an extra communication step to the normal path of PBFT in exchange for reducing the communication complexity of leader change. Due to space constraints, we defer the details about these two protocols to [17, §D] and [17, §E]. Their liveness proofs follow the methodology we proposed for PBFT-light, establishing analogs of Lemmas 4-6.

For PBFT with periodic leader rotation we have also established latency bounds when using the synchronizer in Figure 3 (see [17, §D]). The most interesting one (Theorem 56) demonstrates the benefit of PBFT's mechanism for adapting timeouts to an unknown δ: recall that in PBFT a process only increases its timeouts when a timer expires, which means that the current view does not operate normally (§4). We show that, since the protocol does

not increase its timeouts in good views (with correct leaders and under synchrony), it pays a minimal latency penalty to recover the first time it encounters a bad leader – the initial value of dur_recovery. This contrasts with the simplistic way of adapting the timeouts to an unknown δ by increasing them in every view: in this case, as the protocol keeps changing views, the timeouts would eventually increase up to the maximum (determined by Δ), and the protocol would have to wait that much to recover from a faulty leader.

8 Related Work and Discussion

Failure detectors. Failure detectors and leader oracles [26, 34] have been widely used for implementing consensus and SMR under benign failures [38,39,47], but their implementations under Byzantine failures are either impractical [43] or detect only restricted failure types [30,40, 46]. Another approach was proposed in a textbook by Cachin et al. [20]. This relies on a leader-based Byzantine Epoch-Change (BEC) abstraction, which accepts "complain" hints from the application suggesting that the trust in the current leader should be revoked. However, like the classical leader oracles, BEC requires all correct processes to eventually trust the same correct leader, which is impossible to achieve in Byzantine settings. In fact, the BEC-based Byzantine consensus algorithm in §5.6.4 of [20] suffers from a liveness bug, which we describe in [17, §F]. The bug has been confirmed with one of the textbook's authors [19].

Although our `advance` is similar to "complain", we use it to implement a weaker abstraction of an SMR synchronizer. We then obtain properties similar to accuracy and completeness of failure detectors by carefully combining SMR-level timers with uses of `advance` (Lemmas 4-5). Also, while [20] does not specify constraints on the use of "complain" (see [17, §F]), we give a complete characterization of `advance` and show its sufficiency for solving SMR.

BFT-SMaRt [13, 54] built on the ideas of [20] to propose an abstraction of *validated and provable (VP) consensus*, which allows its clients to control leader changes. Although the overall BFT-SMaRt protocol appears to be correct, its liveness proof sketch suffers from issues with rigor similar to those of [20]. In particular, the conditions on how to change the leader in VP-Consensus to ensure its liveness were underspecified (again, see [17, §F]).

Emulating synchrony. Alternative abstractions avoid dependency on the specifics of a failure model by simulating synchrony [14, 27, 35, 42]. The first such abstraction is due to Awerbuch [10] who proposed a family of synchronizer algorithms emulating a round-based synchronous system of top of an asynchronous network with reliable communication and processes. The first such emulation in a failure-prone partially synchronous system was introduced in the DLS paper [32]. It relied on an expensive clock synchronization protocol, which interleaved its messages with every step of a high-level consensus algorithm implemented on top of it. Later work proposed more practical solutions, which reduce the synchronization frequency by relying on either timers [31] or synchronized hardware clocks [3, 7, 36] (the latter can be obtained using one of the existing fault-tolerant clock synchronization algorithms [29,53]). However, the DLS model emulates communication-closed rounds, i.e., eventually, a process in a round r receives *all* messages sent by correct processes in r. This property rules out *optimistically responsive* [51,57] protocols such as PBFT, which can make progress as soon as they receive messages from *any quorum*.

Consensus synchronizers. To address the shortcoming of DLS rounds, recent work proposed a more flexible abstraction ("consensus synchronizer" in §3) that switches processes through an infinite series of *views* [16, 49, 57]. In contrast to rounds, each view may subsume multiple

communication steps. Although consensus synchronizers can be used for efficient single-shot Byzantine consensus [16], using them for SMR results in suboptimal implementations. A classical approach is to decide on each SMR command using a separate black-box consensus instance [52]. However, implementing the latter using a consensus synchronizer would force the processes in every instance to iterate over the same sequence of potentially bad views until the one with a correct leader and sufficiently long duration could be reached.

An alternative approach was proposed in HotStuff [57]. This SMR protocol is driven by a *pacemaker*, which keeps generating views similarly to a consensus synchronizer. Within each view HotStuff runs a voting protocol that commits a block of client commands in a growing hash chain. Although the voting protocol is optimistically responsive, committing the next block is delayed until the pacemaker generates a new view, which increases latency. The cost the pacemaker may incur to generate a view is also paid for every single block.

SMR synchronizers. In contrast to the above approaches, SMR synchronizers allow the application to initiate view changes on demand via an `advance` call. As we show, this affords SMR protocols the flexibility to judiciously manage their view synchronization schedule: in particular, it prevents the timeouts from growing unnecessarily (§7) and avoids the overheads of further view synchronizations once a stable view is reached (Lemma 5, §5).

The first synchronizer with a `new_view`/`advance` interface, which here we call an SMR synchronizer, was proposed by Naor et al. [48,49]. They used it as an intermediate module in a communication-efficient implementation of a consensus synchronizer. The latter is sufficient to ensure the liveness of HotStuff [57] via either of the two straightforward SMR constructions we described above. The specification of the `new_view`/`advance` module of Naor et al. was only used as a stepping stone in the proof of their consensus synchronizer, and as a result, is more low-level and complex than our SMR synchronizer specification. Naor et al. did not investigate the usability of the SMR synchronizer abstraction as a generic building block applicable to a wide range of Byzantine SMR protocols – a gap we fill in this paper. Finally, they only handled a simplified version of partial synchrony where messages are never lost and δ is known a priori, whereas our SMR synchronizer implementation handles partial synchrony in its full generality. This implementation builds on the consensus synchronizer of Bravo et al. [16]. However, its correctness proof and performance analysis are more intricate, since the timing of the view switches is not fixed a priori, but driven by external `advance` inputs.

Aştefănoaei et al. [6] proposed another framework for implementing Byzantine SMR protocols, based on DLS rounds. This uses a simple synchronizer that does not exchange any messages: it recovers from a period of asynchrony by progressively increasing round durations until they are long enough for all correct processes to overlap in the same round. This way of view synchronization rules out optimistically responsive SMR protocols and does not bound the time to reach a decision after GST, as we do.

SMR liveness proofs. PBFT [23–25] is a seminal protocol whose design choices have been widely adopted [37,44,55,56]. To the best of our knowledge, our proof in §5 is the first one to formally establish its liveness. An informal argument given in [25, §4.5.1] mainly justifies liveness assuming all correct processes enter a view with a correct leader and stay in that view for sufficiently long. It does not rigorously justify why such a view will be eventually reached, and in particular, how this is ensured by the interplay between SMR-level timeout management and view synchronization (§5). Liveness mechanisms were also omitted from the formal specification of PBFT by an I/O-automaton [23,25].

Bravo et al. [16] have applied consensus synchronizers to several consensus protocols, including a single-shot version of PBFT. These protocols and their proofs and are much more straightforward than the full SMR protocols we consider here. In particular, since a consensus synchronizer keeps switching processes between views regardless of whether their leaders are correct, the proof of the single-shot PBFT in [16] does not need to establish analogs of completeness and accuracy (Lemmas 4 and 5) or deal with the fact that processes may disagree on timeout durations (Lemma 6).

Byzantine SMR protocols often integrate view synchronization into the core protocol, enabling white-box optimizations [1, 9, 18, 24]. Our work does not rule out this approach, but allows making it more systematic: we can first develop efficient mechanisms for view synchronization independently from SMR protocols, and do white-box optimizations afterwards.

References

1. DiemBFT v4: State machine replication in the Diem blockchain. URL: `https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf`.
2. Incorrect by construction-CBC Casper isn't live. URL: `https://derekhsorensen.com/docs/CBC_Casper_Flaw.pdf`.
3. Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous Byzantine agreement with expected $O(1)$ rounds, expected $O(n^2)$ communication, and optimal resilience. In *Conference on Financial Cryptography and Data Security (FC)*, 2019.
4. Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Ramakrishna Kotla, and Jean-Philippe Martin. Revisiting fast practical Byzantine fault tolerance. *arXiv*, abs/1712.01367, 2017. `arXiv:1712.01367`.
5. Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of Byzantine broadcast: a complete categorization. In *Symposium on Principles of Distributed Computing (PODC)*, 2021.
6. Lăcrămioara Aştefănoaei, Pierre Chambart, Antonella Del Pozzo, Thibault Rieutord, Sara Tucci-Piergiovanni, and Eugen Zălinescu. Tenderbake – A solution to dynamic repeated consensus for blockchains. In *Symposium on Foundations and Applications of Blockchain (FAB)*, 2021.
7. Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. Generating fast indulgent algorithms. In *International Conference on Distributed Computing and Networking (ICDCN)*, 2011.
8. Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. Correctness of Tendermint-core blockchains. In *Conference on Principles of Distributed Systems (OPODIS)*, 2018.
9. Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. Dissecting Tendermint. In *Conference on Networked Systems (NETYS)*, 2019.
10. Baruch Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, 1985.
11. Rida A. Bazzi and Yin Ding. Non-skipping timestamps for Byzantine data storage systems. In *Symposium on Distributed Computing (DISC)*, 2004.
12. Christian Berger, Hans P. Reiser, and Alysson Bessani. Making reads in BFT state machine replication fast, linearizable, and live. In *Symposium on Reliable Distributed Systems (SRDS)*, 2021.
13. Alysson Neves Bessani, João Sousa, and Eduardo Adílio Pelinson Alchieri. State machine replication for the masses with BFT-SMART. In *Conference on Dependable Systems and Networks (DSN)*, 2014.
14. Martin Biely, Josef Widder, Bernadette Charron-Bost, Antoine Gaillard, Martin Hutle, and André Schiper. Tolerating corrupted communication. In *Symposium on Principles of Distributed Computing (PODC)*, 2007.

15 Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.

16 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making Byzantine consensus live. In *Symposium on Distributed Computing (DISC)*, 2020.

17 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Liveness and latency of Byzantine state-machine replication (extended version). *arXiv*, abs/2202.06679, 2022. arXiv:2202.06679.

18 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *arXiv*, abs/1807.04938, 2018. arXiv:1807.04938.

19 Christian Cachin. Personal communication, 2022.

20 Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2 ed.)*. Springer, 2011.

21 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *International Cryptology Conference (CRYPTO)*, 2001.

22 Christian Cachin and Marko Vukolić. Blockchain consensus protocols in the wild (keynote talk). In *Symposium on Distributed Computing (DISC)*, 2017.

23 Miguel Castro. *Practical Byzantine Fault Tolerance*. PhD thesis, Massachusetts Institute of Technology, 2001.

24 Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

25 Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

26 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

27 Bernadette Charron-Bost and André Schiper. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Comput.*, 22(1):49–71, 2009.

28 Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

29 Danny Dolev, Joseph Y. Halpern, Barbara Simons, and Ray Strong. Dynamic fault-tolerant clock synchronization. *J. ACM*, 42(1):143–185, 1995.

30 Assia Doudou, Benoît Garbinato, and Rachid Guerraoui. Abstractions for devising Byzantine-resilient state machine replication. In *Symposium on Reliable Distributed Systems (SRDS)*, 2000.

31 Cezara Dragoi, Josef Widder, and Damien Zufferey. Programming at the edge of synchrony. *Proc. ACM Program. Lang.*, 4(OOPSLA), 2020.

32 Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

33 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

34 Felix C. Freiling, Rachid Guerraoui, and Petr Kuznetsov. The failure detector abstraction. *ACM Comput. Surv.*, 43(2):9:1–9:40, 2011.

35 Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony. In *Symposium on Principles of Distributed Computing (PODC)*, 1998.

36 Seth Gilbert, Rachid Guerraoui, and Dariusz R. Kowalski. On the message complexity of indulgent consensus. In *Symposium on Distributed Computing (DISC)*, 2007.

37 Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A scalable and decentralized trust infrastructure. In *Conference on Dependable Systems and Networks (DSN)*, 2019.

38 Rachid Guerraoui. Indulgent algorithms (preliminary version). In *Symposium on Principles of Distributed Computing (PODC)*, 2000.

39 Rachid Guerraoui and Michel Raynal. The information structure of indulgent consensus. *IEEE Transactions on Computers*, 53(4):453–466, 2004.

40 Andreas Haeberlen and Petr Kuznetsov. The fault detection problem. In *Conference on Principles of Distributed Systems (OPODIS)*, 2009.

41 Amir Herzberg and Shay Kutten. Fast isolation of arbitrary forwarding faults. In *Symposium on Principles of Distributed Computing (PODC)*, 1989.

42 Idit Keidar and Alexander Shraer. Timeliness, failure-detectors, and consensus performance. In *Symposium on Principles of Distributed Computing (PODC)*, 2006.

43 Kim Potter Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35, 2003.

44 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4):7:1–7:39, 2010.

45 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

46 Dahlia Malkhi and Michael Reiter. Unreliable intrusion detection in distributed computations. In *Workshop on Computer Security Foundations (CSFW)*, 1997.

47 Achour Mostéfaoui and Michel Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors: A general quorum-based approach. In *Symposium on Distributed Computing (DISC)*, 1999.

48 Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. In *Cryptoeconomics Systems Conference (CES)*, 2020.

49 Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear Byzantine SMR. In *Symposium on Distributed Computing (DISC)*, 2020.

50 Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. In *Symposium on Distributed Computing (DISC)*, 2017.

51 Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2018.

52 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

53 Barbara Simons, Jennifer Welch, and Nancy Lynch. An overview of clock synchronization. In *Fault-Tolerant Distributed Computing*, 1986.

54 João Sousa. *Byzantine State Machine Replication for the Masses*. PhD thesis, University of Lisbon, 2017.

55 Chrysoula Stathakopoulou, Tudor David, and Marko Vukolić. Mir-BFT: High-throughput BFT for blockchains. *arXiv*, abs/1906.05552, 2019. `arXiv:1906.05552`.

56 Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Symposium on Reliable Distributed Systems (SRDS)*, 2009.

57 Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Symposium on Principles of Distributed Computing (PODC)*, 2019.

Oracular Byzantine Reliable Broadcast

Martina Camaioni
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Rachid Guerraoui
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Matteo Monti
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Manuel Vidigueira
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

Byzantine Reliable Broadcast (BRB) is a fundamental distributed computing primitive, with applications ranging from notifications to asynchronous payment systems. Motivated by practical consideration, we study Client-Server Byzantine Reliable Broadcast (CSB), a multi-shot variant of BRB whose interface is split between broadcasting *clients* and delivering *servers*. We present Draft, an optimally resilient implementation of CSB. Like most implementations of BRB, Draft guarantees both liveness and safety in an asynchronous environment. Under good conditions, however, Draft achieves unparalleled efficiency. In a moment of synchrony, free from Byzantine misbehaviour, and at the limit of infinitely many broadcasting clients, a Draft server delivers a b-bits payload at an asymptotic amortized cost of 0 signature verifications, and $(log_2(c) + b)$ bits exchanged, where c is the number of clients in the system. This is the information-theoretical minimum number of bits required to convey the payload (b bits, assuming it is compressed), along with an identifier for its sender ($\log_2(c)$ bits, necessary to enumerate any set of c elements, and optimal if broadcasting frequencies are uniform or unknown). These two achievements have profound practical implications. Real-world BRB implementations are often bottlenecked either by expensive signature verifications, or by communication overhead. For Draft, instead, the network is the limit: a server can deliver payloads as quickly as it would receive them from an infallible oracle.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Byzantine reliable broadcast, Good-case complexity, Amortized complexity, Batching

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.13

Related Version The full version of this paper, which includes all detailed proofs and pseudocode, is available online.
Full Version: `https://arxiv.org/abs/` [17]

1 Introduction

Byzantine reliable broadcast (BRB) is one of the most fundamental and versatile building blocks in distributed computing, powering a variety of Byzantine fault-tolerant (BFT) systems [14, 28]. The BRB abstraction has recently been shown to be strong enough to process payments, enabling cryptocurrency deployments in an asynchronous environment [29]. Originally introduced by Bracha [9] to allow a set of processes to agree on a single message from a designated sender, BRB naturally generalizes to the multi-shot case, enabling higher-level abstractions such as Byzantine FIFO [44, 12] and causal [7, 4] broadcast. We study a practical, multi-shot variant of BRB whose interface is split between broadcasting *clients* and delivering *servers*. We call this abstraction Client-Server Byzantine Reliable Broadcast (CSB).

© Martina Camaioni, Rachid Guerraoui, Matteo Monti, and Manuel Vidigueira;
licensed under Creative Commons License CC-BY 4.0

CSB in brief. Clients broadcast, and servers deliver, *payloads* composed by a *context* and a *message*. This interface allows, for example, Alice to announce her wedding as well as will her fortune by respectively broadcasting

$$\Big(\underbrace{\texttt{"My wife is"}}_{\text{context } c_w}, \underbrace{\texttt{"Carla"}}_{\text{message } m_w}\Big) \quad \Big(\underbrace{\texttt{"All my riches go to"}}_{\text{context } c_r}, \underbrace{\texttt{"Bob"}}_{\text{message } m_r}\Big)$$

CSB guarantees that: (*Consistency*) no two correct servers deliver different messages for the same client and context; (*Totality*) either all correct servers deliver a message for a given client and context, or no correct server does; (*Integrity*) if a correct server delivers a payload from a correct client, then the client has broadcast that payload; and (*Validity*) a payload broadcast by a correct client is delivered by at least one correct server. Following from the above example, Carla being Alice's wife does not conflict with Bob being her sole heir (indeed, $c_w \neq c_r$), but Alice would not be able to convince two correct servers that she married Carla and Diana, respectively. Higher-level broadcast abstractions can be easily built on top of CSB. For example, using integer sequence numbers as contexts and adding a reordering layer yields Client-Server Byzantine FIFO Broadcast. For the sake of CSB, however, it is not important for contexts to be integers, or satisfy any property other than comparability. Throughout the remainder of this paper, the reader can picture contexts as opaque binary blobs. Lastly, while the set of servers is known, CSB as presented does not assume any client to be known a priori. The set of clients can be permissionless, with servers discovering new clients throughout the execution.

A utopian model. Real-world BRB implementations are often bottlenecked either by expensive signature verifications [21] or by communication overhead [10, 34, 35]. With the goal of broadening those bottlenecks, simplified, more trustful models are useful to establish a (sometimes grossly unreachable) bound on the efficiency that an algorithm can attain in the Byzantine setting. For example, in a utopian model where any agreed-upon process can be trusted to never fail (let us call it an *oracle*), CSB can easily be implemented with great efficiency. Upon initialization, the oracle organizes all clients in a list, which it disseminates to all servers. For simplicity, let us call *id* a client's position in the list. To broadcast a payload p, a client with id i simply sends p to the oracle: the oracle checks p for equivocation (thus ensuring consistency), then forwards (i, p) to all servers (thus ensuring validity and totality). Upon receiving (i, p), a server blindly trusts the oracle to uphold all CSB properties, and delivers (i, p). Oracle-CSB is clearly very efficient. On the one hand, because the oracle can be trusted not to attribute spurious payloads to correct clients, integrity can be guaranteed without any server-side signature verification. On the other, in order to deliver (i, p), a server needs to receive just $(\lceil \log_2(c) \rceil + |p|)$ bits, where c denotes the total number of clients, and $|p|$ measures p's length in bits. This is optimal assuming the rate at which clients broadcast is unknown[1] or uniform[2] [20].

Matching the oracle. Due to its reliance on a single infallible process, Oracle-CSB is not a fault-tolerant distributed algorithm: shifting back to the Byzantine setting, a single failure would be sufficient to compromise all CSB properties. Common sense suggests that Byzantine

[1] Lacking an assumption on broadcasting rates, an adversarial scheduler could have all messages broadcast by the client with the longest id, which we cannot guarantee to be shorter than $\lceil \log_2(c) \rceil$ bits.

[2] Should some clients be expected to broadcast more frequently than others, we could further optimize Oracle-CSB by assigning smaller ids to more active clients, possibly at the cost of having less active clients have ids whose length exceeds $\lceil \log_2(c) \rceil$. Doing so, however, is beyond the scope of this paper.

resilience will necessarily come at some cost: protocol messages must be exchanged to preserve consistency and totality, signatures must be produced and verified to uphold integrity and, lacking the totally-ordering power that only consensus can provide, ids cannot be assigned in an optimally dense way. However, this paper proves the counter-intuitive result that an asynchronous, optimally-resilient, Byzantine implementation of CSB can asymptotically match the efficiency of Oracle-CSB. This is not just up to a constant, but identically. In a synchronous execution, free from Byzantine misbehaviour, and as the number of concurrently broadcasting clients goes to infinity (we call these conditions the *batching limit*[3]), our CSB implementation Draft delivers a payload p at an asymptotic[4], amortized cost of 0 signature verifications[5] and $(\lceil \log_2(c) \rceil + |p|)$ bits exchanged per server, the same as in Oracle-CSB (we say that Draft achieves *oracular efficiency*). At the batching limit a Draft server is dispensed from nearly all signature verifications, as well as nearly all traffic that would be normally required to convey protocol messages, signatures, or client public keys. Network is the limit: payloads are delivered as quickly as they can be received.

CSB's common bottlenecks. To achieve oracular efficiency, we focus on three types of server overhead that commonly affect a real-world implementation of CSB:

- *Protocol overhead.* Safekeeping consistency and totality typically requires some form of communication among servers. This communication can be direct (as in Bracha's original, all-to-all BRB implementation) or happen through an intermediary (as in Bracha's signed, one-to-all-to-one BRB variant), usually employing signatures to establish authenticated, intra-server communication channels through a (potentially Byzantine) relay.
- *Signature overhead.* Upholding integrity usually requires clients to authenticate their messages using signatures. For servers, this entails both a computation and a communication overhead. On the one hand, even using well-optimized schemes, signature verification is often CPU-heavy enough to dominate a server's computational budget, dwarfing in particular the CPU footprint of much lighter, symmetric cryptographic primitives such as hashes and ciphers. On the other hand, transmitting signatures results in a fixed communication overhead per payload delivered. While the size of a signature usually ranges from a few tens to a few hundreds of bytes, this overhead is non-negligible in a context where many clients broadcast small messages. This is especially true in the case of payments, where a message reduces to the identifier of a target account and an integer to denote the amount of money to transfer.
- *Identifier overhead.* CSB's multi-shot nature calls for a sender identifier to be attached to each broadcast payload. Classically, the client's public key is used as identifier. This is convenient for two reasons. First, knowing a client's identifier is sufficient to authenticate its payloads. Second, asymmetric keypairs have very low probability of collision. As such, clients can create identities in the system without any need for coordination: locally generating a keypair is sufficient to begin broadcasting messages. By cryptographic design, however, public keys are sparse, and their size does not change with the number of clients. This translates to tens to hundreds of bytes being invested to identify a client from a set that can realistically be enumerated by a few tens of bits. Again, this communication overhead is heavier on systems where broadcasts are frequent and brief.

[3] The batching limit includes other easily achievable, more technical conditions that we omit in this section for the sake of brevity. For the full definition, please refer to the extended version of this paper [17].

[4] The asymptotic costs are reached quite fast, at rates comparable to C^{-1} or $log(C) \cdot C^{-1}$.

[5] This does not mean that batches are processed in constant time: hashes and signature aggregations, for example, still scale linearly in the size of a batch. The real-world computational cost of such simple operations, however, is several orders of magnitude lower than that of signature verification.

On the way to matching Oracle-CSB's performance, we develop techniques to negate all three types of overhead: at the batching limit, a Draft server delivers a payload wasting 0 bits to protocol overhead, performing 0 signature verifications, and exchanging $\lceil \log_2(c) \rceil$ bits of identifier, the minimum required to enumerate the set of clients. We outline our contributions below, organized in three (plus one) take-home messages (T-HMs).

T-HM1: The effectiveness of batching goes beyond total order. In the totally ordered setting, batching is famously effective at amortizing protocol overhead [45, 3]. Instead of disseminating its message to all servers, a client hands it over to (one or more)[6] batching processes. Upon collecting a large enough set of messages, a batching process organizes all messages in a batch, which it then disseminates to the servers. Having done so, the batching process submits the batch's hash to the system's totally-ordering primitive. Because hashes are constant in length, the cost of totally ordering a batch does not depend on its size. Once batches are totally ordered, so too are messages (messages within a batch can be ordered by any deterministic function), and equivocations can be handled at the application layer (for example, in the context of a cryptocurrency, the second request to transfer the same asset can be ignored by all correct servers, with no need for additional coordination). At the limit of infinitely large batches, the relative overhead of the ordering protocol becomes vanishingly small, and a server can allocate virtually all of its bandwidth to receiving batches. This strategy, however, does not naturally generalize to CSB, where batches lack total order. As payloads from multiple clients are bundled in the same batch, a correct server might detect equivocation for only a subset of the payloads in the batch. Entirely accepting or entirely rejecting a partially equivocated batch is not an option. In the first case, consistency could be violated. In the second case, a single Byzantine client could single-handedly "poison" the batches assembled by every correct batching process with equivocated payloads, thus violating validity. In Draft, a server can partially reject a batch, acknowledging all but some of its payloads. Along with its partial acknowledgement, a server provides a proof of equivocation to justify each exception. Having collected a quorum of appropriately justified partial acknowledgements, a batching process has servers deliver only those payloads that were not excepted by any server. Because proofs of equivocations cannot be forged for correct clients, a correct client handing over its payload to a correct batching process is guaranteed to have that payload delivered. In the common case where batches have little to no equivocations, servers exchange either empty or small lists of exceptions, whose size does not scale with that of the batch. This extends the protocol-amortizing power of batching to CSB and, we conjecture, other non-totally ordered abstractions.

T-HM2: Interactive multi-signing can slash signature overhead. Traditionally, batching protocols are non-interactive on the side of clients. Having offloaded its message to a correct batching process, a correct client does not need to interact further for its message to be delivered: the batching process collects an arbitrary set of independently signed messages and turns to the servers to get each signature verified, and the batch delivered. This approach is versatile (messages are not tied to the batch they belong to) and reliable (a client crashing does not affect a batch's progress) but expensive (the cost of verifying each signature is high and independent of the batch's size). In Draft, batching processes engage in an interactive protocol with clients to replace, in the good case, all individual signatures in a batch with

[6] In most real-world implementations, a client optimistically entrusts its payload to a single process, extending its request to larger portions of the system upon expiration of a suitable timeout.

a single, batch-wide *multi-signature*. In brief, multi-signature schemes extend traditional signatures with a mechanism to *aggregate* signatures and public keys: an arbitrarily large set of signatures for the same message[7] can be aggregated into a single, constant-sized signature; similarly, a set of public keys can be aggregated into a single, constant-sized public key. The aggregation of a set of signatures can be verified in constant time against the aggregation of all corresponding public keys. Unlike verification, aggregation is a cheap operation, reducing in some schemes to a single multiplication on a suitable field. Multi-signature schemes open a possibility to turn expensive signature verification into a once-per-batch operation. Intuitively, if each client contributing to a batch could multi-sign the entire batch instead of its individual payload, all multi-signatures could be aggregated, allowing servers to authenticate all payloads at once. However, as clients cannot predict how their payloads will be batched, this must be achieved by means of an interactive protocol. Having collected a set of individually-signed payloads in a batch, a Draft batching process shows to each contributing client that its payload was included in the batch. In response, clients produce their multi-signatures for the batch's hash, which the batching process aggregates. Clients that fail to engage in this interactive protocol (e.g., because they are faulty or slow) do not lose liveness, as their original signature can still be attached to the batch to authenticate their individual payload. In the good case, all clients reply in a timely fashion, and each server has to verify a single multi-signature per batch. At the limit of infinitely large batches, this results in each payload being delivered at an amortized cost of 0 signature verifications. The usefulness of this interactive protocol naturally extends beyond CSB to all multi-shot broadcast abstractions whose properties include integrity.

T-HM3: Dense id assignment can be achieved without consensus. In order to efficiently convey payload senders, Oracle-CSB's oracle organizes all clients in a list, attaching to each client a successive integral identifier. Once the list is disseminated to all servers, the oracle can identify each client by its identifier, sparing servers the cost of receiving larger, more sparse, client-generated public keys. Id-assignment strategies similar to that of Oracle-CSB can be developed, in the distributed setting, building on top of classical algorithms that identify clients by their full public keys (we call such algorithms *id-free*, as opposed to algorithms such as Draft, which are *id-optimized*). In a setting where consensus can be achieved, the identifier density of Oracle-CSB is easily matched. Upon initialization, each client submits its public key to an id-free implementation of Total-Order Broadcast (TOB). Upon delivery of a public key, every correct process agrees on its position within the common, totally-ordered log. As in Oracle-CSB, each client can then use its position in the list as identifier within some faster, id-optimized broadcast implementation. In a consensus-less setting, achieving a totally-ordered list of public keys is famously impossible [26]. This paper, however, proves the counter-intuitive result that, when batching is used, the density of ids assigned by a consensus-less abstraction can asymptotically match that of those produced by Oracle-CSB or consensus. In Dibs, our consensus-less id-assigning algorithm, a client requests an id from every server. Each server uses an id-free implementation of FIFO Broadcast to order the client's public key within its own log. Having observed its public key appear in at least one log, the client publicly elects the server in charge of that log to be its *assigner*. Having done so, the client obtains an id composed of the assigner's public key and the client's position

[7] Some multi-signature schemes also allow the aggregation of signatures on heterogeneous messages. In that case, however, aggregation is usually as expensive as signature verification. Given our goal to reduce CPU complexity for servers, this paper entirely disregards heterogeneous aggregation schemes.

within the assigner's log. We call the two components of an id *domain* and *index*, respectively. Because the set of servers is known to (and can be enumerated by) all processes, an id's domain can be represented in $\lceil \log_2(n) \rceil$ bits, where n denotes the total number of servers. Because at most c distinct clients can appear in the FIFO log of any server, indices are at most $\lceil \log_2(c) \rceil$ bits long. In summary, Dibs assigns ids to clients without consensus, at an additional cost of $\lceil \log_2(n) \rceil$ bits per id. Interestingly, even this additional complexity can be amortized by batching. Having assembled a batch, a Draft batching process represents senders not as a list of ids, but as a map, associating to each of the n domains the indices of all ids in the batch under that domain. At the limit of infinitely large batches ($C \gg N$), the bits required to represent the map's keys are entirely amortized by those required to represent its values. This means that, while $(\lceil \log_2(n) \rceil + \lceil \log_2(c) \rceil)$ bits are required to identify a client in isolation, $\lceil \log_2(c) \rceil$ bits are sufficient if the client is batched: even without consensus, Draft asymptotically matches the id efficiency of Oracle-CSB.

Bonus T-HM: Untrusted processes can carry the system. In THM1, we outlined how batching can be generalized to the consensus-less case, and discussed its role in removing protocol overhead. In THM2, we sketched how an interactive protocol between clients and batching processes can eliminate signature overhead. In employing these techniques, we shifted most of the communication and computation complexity of our algorithms from servers to batching processes. Batching processes verify all client signatures, create batches, verify and aggregate all client multi-signatures, then communicate with servers in an expensive one-to-all pattern, engaging server resources (at the batching limit) as little as an oracle would. Our last contribution is to observe that a batching process plays no role in upholding CSB's safety. As we discuss in detail throughout the remainder of this paper, a malicious batching process cannot compromise consistency (it would need to collect two conflicting quorums of acknowledgements), totality (any server delivering a batch has enough information to convince all others to do the same) or integrity (batches are still signed, and forged or improperly aggregated multi-signatures are guaranteed to be detected). Intuitively, the only damage a batching process can do to the system is to refuse to process client payloads[8]. This means that a batching process does not need to satisfy the same security properties as a server. CSB's properties cannot be upheld if a third of the servers are faulty. Conversely, Draft has both liveness and safety as long as *a single* batching process is correct. This observation has profound practical implications. In the real world, scaling the resources of a permissioned, security-critical set of servers can be hard. On the one hand, reputable, dependable institutions partaking in the system might not have the resources to keep up with its demands. On the other, more trusted hardware translates to a larger security cross-section. Trustless processes, however, are plentiful to the point that permissionless cryptocurrencies traditionally waste their resources, making them compete against each other in expensive proofs of Sybil-resistance [39]. In this paper, we extend the classical client-server model with *brokers*, a permissionless, scalable set of processes whose only purpose is to alleviate server complexity. Unlike servers, more than two-thirds of which we assume to be correct, all brokers but one can be faulty. In Draft, brokers act as an intermediary between clients and servers, taking upon themselves the batching of payloads, verification and aggregation of signatures, the dissemination of batches, and the transmission of protocol messages.

[8] Or cause servers to waste resources, e.g., by transmitting improperly signed batches. Simple accountability measures, we conjecture, would be sufficient to mitigate these attacks in Draft. A full discussion of Denial of Service, however, is beyond the scope of this paper.

Roadmap. We discuss related work in Section 2. We state our model and recall useful cryptographic background in Section 3. In Section 4, we introduce our CSB implementation Draft: we overview Draft's protocol in Section 4.1, and provide high-level arguments for Draft's efficiency in Section 4.2. We draw our conclusions and propose future work in Section 5. The full formal analysis of our algorithms as well as their pseudocode can be found in the extended version of this paper [17].

2 Related Work

Byzantine Reliable Broadcast (BRB) is a classical primitive of distributed computing, with widespread practical applications such as in State Machine Replication (SMR) [38, 15, 11], Byzantine agreement [40, 18, 32, 31, 47], blockchains [3, 22, 23], and online payments [29, 19, 33]. In classical BRB, a system of n processes agree on a single message from a single *source* (one of the n processes), while tolerating up to f Byzantine failures (f of the n processes can behave arbitrarily). A well known solution to asynchronous BRB with provably optimal resilience ($f < n/3$) was first proposed by Bracha [8, 9] who introduced the problem. Bracha's broadcast reaches $O(n^2)$ message complexity, and $O(n^2L)$ communication complexity (total number of transmitted bits between correct processes [48]), where L is the length of the message. Since $O(n^2)$ message complexity is provably optimal [27], the main focus of BRB-related research has been on reducing its communication complexity. The best lower bound for communication complexity is $\Omega(nL + n^2)$, although it is unknown whether it is tight. The nL term comes from all processes having to receive the message (length L), while the n^2 term comes from each of the n processes having to receive $\Omega(n)$ protocol messages to ensure agreement in the presence of $f = \Theta(n)$ failures [27]. One line of research focuses on worst-case complexity, predominantly using error correcting codes [43, 6] or erasure codes [41, 30, 16, 2], and has produced various BRB protocols with improved complexity [2, 16, 13, 24, 40], many of them quite recently. The work of Das, Xiang and Ren [24] achieves $O(nL + kn^2)$ communication complexity (specifically, $7nL + 2kn^2$), where k is the security parameter (e.g., the length of a hash, typically 256 bits). As the authors note, the value of hidden constants (and k, which is sometimes considered as a constant in literature) is particularly important when considering practical implementations of these protocols. Another line of research focuses on optimizing the good case performance of BRB, i.e., when the network behaves synchronously and no process misbehaves [13, 18, 32, 42, 1]. As the good case is usually the common case, in practice, the real-world communication complexity of these optimistic protocols matches that of the good case. A simple and widely-used hash-based BRB protocol is given by Cachin *et al.* [13]. It replaces the echo and ready phase messages in Bracha's protocol with hashes, achieving $O(nL + kn^2)$ in the good case (specifically, $nL + 2kn^2$), and $O(n^2L)$ in the worst-case. Considering practical throughput, some protocols also focus on the *amortized* complexity per source message [18, 42, 36]. Combining techniques such as *batching* [18] and threshold signatures [46], at the limit (of batch size), BRB protocols reach $O(nL)$ amortized communication complexity in the good case [42]. At this point, the remaining problem lies in the hidden constants. In the authenticated setting, batching-based protocols rely on digital signatures to validate (source) messages before agreeing to deliver them [42]. In reality, each source message in a batch includes its content, an identifier of the source (e.g., a k-sized public key), a sequence id (identifying the message), and a k-sized signature. When considering systems where L is small (e.g., online payments), these can take up a large fraction of the communication. To be precise, the good-case amortized communication complexity would be $O(nL + kn)$. In fact, message signatures (the kn factor)

are by far the main bottleneck in practical applications of BRB today [23, 47], both in terms of communication and computation (signature verification), leading to various attempts at reducing or amortizing their cost [22, 36]. For example, Crain *et al.* [22] propose *verification sharding*, in which only $f+1$ processes have to receive and verify all message signatures in the good case, which is a 3-fold improvement over previous systems (on the kn factor) where all n processes verify all signatures. However, by itself, this does not improve on the amortized cost of $O(nL+kn)$ per message. When contrasting theoretical research with practical systems, it is interesting to note the gap that can surge between the theoretical model and reality. The recent work of Abraham *et al.* [1], focused on the good-case latency of Byzantine broadcast, expands on some of these mismatches and argues about the practical limitations of focusing on the worst-case. Another apparent mismatch lies in the classical model of Byzantine broadcast. In many of the applications of BRB mentioned previously (e.g., SMR, permissioned blockchains, online payments), there is usually a set of *servers* (n, up to f of which are faulty), and a set of external clients (X) which are the true sources of messages. The usual transformation from BRB's classical model into these practical settings maps the set of n servers as the n processes and simply excludes clients as system entities, e.g., assuming their messages are relayed through one of the servers. Since the number of clients can be very large ($|X| \gg n$), clients are untrusted (which can limit their usefulness), and the focus is on the communication complexity of the *servers*, this transformation seems reasonable and simplifies the problem. However, it can also limit the search for more practical solutions. In this paper, in contrast with the classical model of BRB, we explicitly include the set of clients X in our system while focusing on the communication complexity surrounding the servers (i.e., the bottleneck). Furthermore, we introduce *brokers*, an untrusted set B of processes, only one of which is assumed to be correct, whose goal is to assist servers in their operation. By doing this, we can leverage brokers to achieve a good-case, amortized communication complexity (for servers, information received or sent) of $nL + o(nL)$.

3 Model & background

3.1 Model

System and adversary. We assume an asynchronous message-passing system where the set Π of processes is the distinct union of three sets: **servers** (Σ), **brokers** (B), and **clients** (X). We use $n = |\Sigma|$, $k = |B|$ and $c = |X|$. Any two processes can communicate via reliable, FIFO, point-to-point links (messages are delivered in the order they are sent). Faulty processes are Byzantine, i.e., they may fail arbitrarily. Byzantine processes know each other, and may collude and coordinate their actions. At most f servers are Byzantine, with $n = 3f + 1$. At least one broker is correct. All clients may be faulty. We use Π_C and Π_F to respectively identify the set of correct and faulty processes. The adversary cannot subvert cryptographic primitives (e.g., forge signatures). Servers and brokers[9] are permissioned (every process knows Σ and B), clients are permissionless (no correct process knows X a priori). We call *certificate* a statement signed by either a plurality ($f + 1$) or a quorum ($2f + 1$) of servers. Since every process knows Σ, any process can verify a certificate.

[9] The assumption that brokers are permissioned is made for simplicity, and can be easily relaxed to the requirement that every correct process knows at least one correct broker.

Good case. The algorithms presented in this paper are designed to uphold all their properties in the model above. Draft, however, achieves oracular efficiency only in the *good case*. In the good case, links are synchronous (messages are delivered at most one time unit after they are sent), all processes are correct, and the set of brokers contains only one element. To take advantage of the good case, Draft makes use of timers (which is uncommon for purely asynchronous algorithms). A timer with timeout δ set at time t rings: after time $(t + \delta)$, if the system is synchronous; after time t, otherwise. Intuitively, in the non-synchronous case, timers disregard their timeout entirely, and are guaranteed to ring only eventually.

3.2 Background

Besides commonly used hashes and signatures, the algorithms presented in this paper make use of two less often used cryptographic primitives, namely, multi-signatures and Merkle trees. We briefly outline their use below. An in-depth discussion of their inner workings, however is beyond the scope of this paper.

Multi-signatures. Like traditional signatures, multi-signatures [5] are used to publicly authenticate messages: a public / secret keypair (p, r) is generated locally; r is used to produce a signature s for a message m; s is publicly verified against p and m. Unlike traditional signatures, however, multi-signatures for the same message can be *aggregated*. Let $(p_1, r_1), \ldots, (p_n, r_n)$ be a set of keypairs, let m be a message, and let s_i be r_i's signature for m. $(p_1, \ldots, p_n)$ and $(s_1, \ldots, s_n)$ can be respectively aggregated into a constant-sized public key $\hat{p}$ and a constant-sized signature $\hat{s}$. As with individually-generated multi-signatures, $\hat{s}$ can be verified in constant time against $\hat{p}$ and m. Aggregation is cheap and non-interactive: provided with $(p_1, \ldots, p_n)$ (resp., $(s_1, \ldots, s_n)$) any process can compute $\hat{p}$ (resp., $\hat{s}$).

Merkle trees. Merkle trees [37] extend traditional hashes with compact *proofs of inclusion*. As with hashes, a sequence $(x_1, \ldots, x_n)$ of values can be hashed into a preimage and collision-resistant digest (or *root*) r. Unlike hashes, however, a proof p_i can be produced from $(x_1, \ldots, x_n)$ to attest that the i-th element of the sequence whose root is r is indeed x_i. In other words, provided with r, p_i and x_i, any process can verify that the i-th element of $(x_1, \ldots, x_n)$ is indeed x_i, without having to learn $(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$. The size of a proof of inclusion for a sequence of n elements is logarithmic in n.

4 Draft: Overview

In this section, we provide an intuitive overview of our CSB implementation, Draft, as well as high-level arguments for its efficiency.

4.1 Protocol

Dramatis personae. The goal of this section is to provide an intuitive understanding of Draft's protocol. In order to do this, we focus on four processes: a correct client χ, a correct broker β, a correct and fast server σ, and a correct but slow server $\tilde{\sigma}$. We follow the messages exchanged between χ, β, σ and $\tilde{\sigma}$ as the protocol unfolds, as captured by Figure 1.

The setting. χ's goal is to broadcast a payload p. χ has already used Draft's underlying Directory abstraction (DIR) to obtain an id i. In brief, DIR guarantees that i is assigned to χ only, and provides χ with an *assignment certificate* a, which χ can use to prove that its id

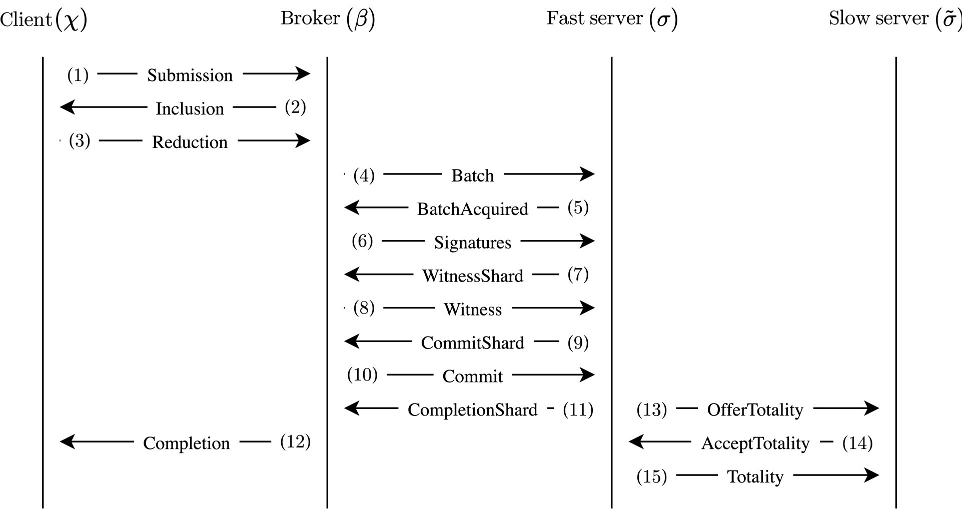


Figure 1 Draft's protocol. Having collected a batch of client payloads, a broker engages in an interactive protocol with clients to reduce the batch, replacing (most of) its individual payload signatures with a single, batch-wide multi-signature. The broker then disseminates the batch to all servers, successively gathering a witness for its correctness and a certificate to commit (some of) its payloads. Having had a plurality of servers deliver the batch, the broker notifies all clients with a suitable certificate. In the bad case, servers can ensure totality without any help from the broker, propagating batches and commit certificates in an all-to-all fashion.

is indeed i. As we discussed in Section 1, Draft uses DIR-assigned ids to identify payload senders. This is essential to Draft's performance, as DIR guarantees *density*: as we outline in Section 4.2, $\lceil \log_2(c) \rceil$ bits are asymptotically sufficient to represent each id in an infinitely large batch. Throughout the remainder of this paper, we say that a process π *knows* an id $\hat{i}$ iff π knows the public keys to which $\hat{i}$ is assigned.

Building a batch. In order to broadcast its payload p, χ produces a signature s for p, and then sends a `Submission` message to β (fig. 1, step 1). The `Submission` message contains p, s, and χ's assignment certificate a. Upon receiving the `Submission` message, β learns χ's id i from a, then verifies s against p. Having done so, β stores (i, p, s) in its *submission pool*. For a configurable amount of time, β fills its pool with submissions from other clients, before *flushing* it into a *batch*. Let us use $(i_1, p_1, s_1), \ldots, (i_b, p_b, s_b)$ to enumerate the elements β flushes from the submission pool (for some n, we clearly have $(i, p, s) = (i_n, p_n, s_n)$). For convenience, we will also use χ_j to identify the sender of p_j (owner of i_j). Importantly, β flushes the pool in such a way that $i_j \neq i_k$ for all $j \neq k$: for safety reasons that will soon be clear, Draft's protocol prevents a client from having more than one payload in any specific batch. Because of this constraint, some payloads might linger in β's pool. This is not an issue: β will simply flush those payloads to a different batch at a later time. When building the batch, β splits submissions and signatures, storing $(i_1, p_1), \ldots, (i_b, p_b)$ separately from $s_1, \ldots, s_b$.

Reducing the batch. Having flushed submissions $(i_1, p_1), \ldots, (i_b, p_b)$ and signatures $s_1, \ldots, s_b$, β moves on to *reduce* the batch, as exemplified in Figure 2. In an attempt to minimize signature overhead for servers, β engages in an interactive protocol with clients $\chi_1, \ldots, \chi_b$ to replace as many signatures as possible with a single, batch-wide multi-signature.

In order to do so, β organizes $(i_1, p_1), \ldots, (i_b, p_b)$ in a Merkle tree with root r (for brevity, we call r the batch's root). β then sends an `Inclusion` message to each χ_j (fig. 1, step 2). Each `Inclusion` message contains r, along with a proof of inclusion q_j for (i_j, p_j). Upon receiving its `Inclusion` message, χ checks q_n against r. In doing so, χ comes to two conclusions. First, χ's submission $(i, p) = (i_n, p_n)$ is part of a batch whose root is r. Second, because no `Draft` batch can contain multiple payloads from the same client, that batch does not attribute χ any payload other than p. In other words, χ can be certain that β will not broadcast some spurious payload $p' \neq p$ in χ's name: should β attempt to do that, the batch would be verifiably malformed, and immediately discarded. This means χ can safely produce a multi-signature m for r: as far as χ is concerned, the batch with root r upholds integrity. Having signed r, χ sends m to β by means of a `Reduction` message (fig. 1, step 3). Upon receiving χ_j's `Reduction` message, β checks χ_j's multi-signature m_j against r. Having done so, β discards χ_j's original signature s_j. Intuitively, with m_j, χ_j attested its agreement with whatever payload the batch attributes to χ_j. Because this is equivalent to individually authenticating p_j, s_j is redundant and can be dropped. Upon expiration of a suitable timeout, β stops collecting `Reduction` messages: clearly, if β waited for every χ_j to produce m_j, a single Byzantine client could prevent the protocol from moving forward by refusing to send its `Reduction` message. β aggregates all the multi-signatures it collected for r into a single, batch-wide multi-signature m. In the good case, every χ_j is correct and timely. If so, β drops all individual signatures, and the entire batch is authenticated by m alone.

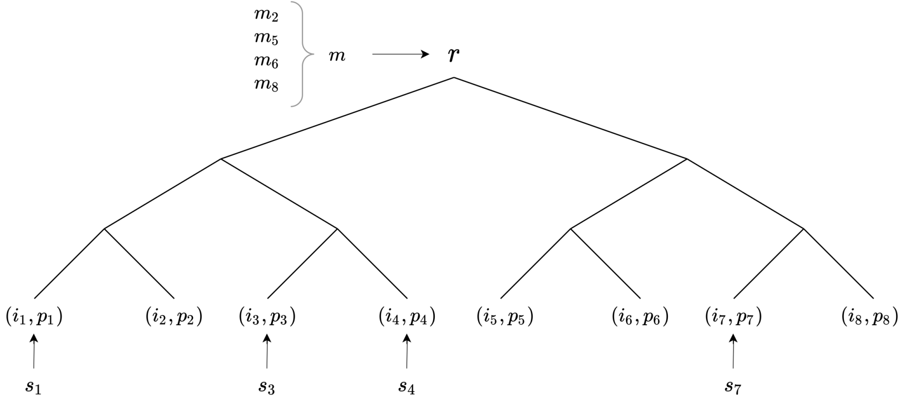


Figure 2 An example of partially reduced batch. $B = 8$ submissions are organized on the leaves of a Merkle tree with root r. Each submission (i_j, p_j) is originally authenticated by an individual signature s_j. Upon collecting a multi-signature m_j for r, the broker drops s_j. Here the broker collected multi-signatures m_2, m_5, m_6 and m_8, leaving a *straggler set* $S = \{(i_1, s_1), (i_3, s_3), (i_4, s_4), (i_7, s_7)\}$. Upon expiration of a suitable timeout, the broker aggregates m_2, m_5, m_6 and m_8 into a single multi-signature m. As such, every payload in the batch is authenticated either by m or by S.

The perks of a reduced batch. Having reduced the batch, β is left with a sequence of submissions $(i_1, p_1), \ldots, (i_b, p_b)$, a multisignature m on the Merkle root r of $(i_1, p_1), \ldots, (i_b, p_b)$, and a *straggler* set S holding the individual signatures that β failed to reduce. More precisely, S contains (i_j, s_j) iff β did not receive a valid `Reduction` message from χ_j before the reduction timeout expired. We recall that m's size is constant, and S is empty in the good case. Once reduced, the batch is cheap to authenticate: it is sufficient to verify the batch's

multi-signature against the batch's root, and each straggler signature against its individual payload. More precisely, let T denote the set of *timely* clients (χ_j is in T iff $(i_j, ..)$ is not in S). Let t denote the aggregation of T's public keys. Provided with $(i_1, p_1), \ldots, (i_b, p_b)$, m and S, any process that knows $i_1, \ldots, i_b$ can verify that the batch upholds integrity by: (1) computing r and t from $(i_1, p_1), \ldots, (i_b, p_b)$ and S; (2) using t to verify m against r; and (3) verifying each s_j in S against p_j. In the good case, authenticating the batch reduces to verifying a single multi-signature. This is regardless of the batch's size.

The pitfalls of a reduced batch. As we discussed in the previous paragraph, reducing a batch makes it cheaper to verify its integrity. Reduction, however, hides a subtle trade-off: once reduced, a batch gets easier to authenticate *as whole*. Its *individual payloads*, however, become harder to authenticate. For the sake of simplicity, let us imagine that β successfully dropped all the individual signatures it originally gathered from $\chi_1, \ldots, \chi_b$. In order to prove that $(\chi = \chi_n)$ broadcast $(p = p_n)$, β could naively produce the batch's root r, (i_n, p_n)'s proof of inclusion q_n, and the batch's multi-signature m for r. This, however, would not be sufficient to authenticate p: because the multi-signature m_n that χ produced for r was aggregated with all others, m can only be verified by the aggregation of *all* $\chi_1, \ldots, \chi_b$'s public keys. This makes authenticating p as expensive as authenticating the entire batch: in order to verify m, *all* (i_j, p_j) must be produced and checked against r, so that all corresponding public keys can be safely aggregated.

Witnessing the batch. As we discuss next, proving the integrity of individual payloads is fundamental to ensure Draft's validity. In brief, to prove that some χ_k equivocated its payload $p_k = (c_k, l_k)$, a server must prove to β that χ_k also issued some payload $p'_k = (c_k, l'_k \neq l_k)$. Lacking this proof, a single Byzantine server could, for example, claim without basis that χ equivocated p. This could trick β into excluding p, thus compromising Draft's validity. As we discussed in the previous paragraph, however, proving the integrity of an individual payload in a reduced batch is difficult. While we conjecture that purely cryptographic solutions to this impasse might be achievable in some schemes[10], Draft has β engage in a simple protocol to further simplify the batch's authentication, replacing all client-issued (multi-)signatures with a single, server-issued certificate. Having collected and reduced the batch, β sends a `Batch` message to all servers (fig. 1, step 4). The `Batch` message only contains $(i_1, p_1), \ldots, (i_b, p_b)$. Upon receiving the `Batch` message, σ collects in a set U_σ all the ids it does not know (i_j is in U_σ iff σ does not know i_j), and sends U_σ back to β by means of a `BatchAcquired` message (fig. 1, step 5). Upon receiving σ's `BatchAcquired` message, β builds a set A_σ containing all id assignments that σ is missing (a_j is in A_σ iff i_j is in U_σ). Having done so, β sends a `Signatures` message to σ (fig. 1, step 6). The `Signatures` message contains the batch's multi-signature m, the straggler set S, and A_σ. We underline the importance of sending id assignments upon request only. Thinking to shave one round-trip off the protocol, β could naively package in a single message all submissions, all (multi-)signatures, and all assignments relevant to the batch. In doing so, however, β would force each server to receive

[10] For example, using BLS, β could aggregate the public keys of $\chi_1, \ldots, \chi_{n-1}, \chi_{n+1}, \ldots, \chi_b$ into a public key $\tilde{t}_n$, then show that the aggregation of $\tilde{t}_n$ with χ's public key correctly verifies m against r. Doing so, however, would additionally require β to exhibit a proof that $\tilde{t}_n$ is not a rogue public key, i.e., that $\tilde{t}_n$ indeed results from the aggregation of client public keys. This could be achieved by additionally having $\chi_1, \ldots, \chi_b$ multi-sign some hard-coded statement to prove that they are not rogues. β could aggregate such signatures on the fly, producing a rogue-resistance proof for $\tilde{t}_n$ that can be transmitted and verified in constant time. This, however, is expensive (and, frankly, at the limit of our cryptographic expertise).

one assignment per submission, immediately forfeiting Draft's oracular efficiency. At the batching limit we assume that all servers already know all broadcasting clients. In that case, both U_σ and A_σ are constant-sized, empty sets, adding only a vanishing amount of communication complexity to the protocol. Upon receiving the `Signatures` message, σ verifies and learns all assignments in A_σ. Having done so, σ knows $i_1, \ldots, i_b$. As we outlined above, σ can now efficiently authenticate the whole batch, verifying m against the batch's root r, and each i_j in S against p_j. Having established the integrity of the whole batch, σ produces a *witness shard* for the batch, i.e., a multi-signature w_σ for $[\texttt{Witness}, r]$, effectively affirming to have successfully authenticated the batch. σ sends w_σ back to β by means of a `WitnessShard` message (fig. 1, step 7). Having received a valid `WitnessShard` message from $f+1$ servers, β aggregates all witness shards into a *witness* w. Because w is a plurality $(f+1)$ certificate, at least one correct server necessarily produced a witness shard for the batch. This means that at least one correct server has successfully authenticated the batch by means of client (multi-)signatures. Because w could not have been gathered if the batch was not properly authenticated, w itself is sufficient to authenticate the batch, and β can drop all (now redundant) client-generated (multi-)signatures for the batch. Unlike m, w is easy to verify, as it is signed by only $f+1$, globally known servers. Like m, w authenticates r. As such, any p_j can now be authenticated just by producing w, and (i_j, p_j)'s proof of inclusion q_j.

Gathering a commit certificate. Having successfully gathered a witness w for the batch, β sends w to all servers by means of a `Witness` message (fig. 1, step 8). Upon receiving the `Witness` message, σ moves on to check $(i_1, p_1), \ldots, (i_b, p_b)$ for equivocations. More precisely, σ builds a set of *exceptions* E_σ containing the ids of all equivocating submissions in the batch (i_j is in E_σ iff σ previously observed χ_j submit a payload p'_j that conflicts with p_j; we recall that p_j and p'_j conflict if their contexts are the same, but their messages are different). σ then produces a *commit shard* for the batch, i.e., a multi-signature c_σ for $[\texttt{Commit}, r, E_\sigma]$, effectively affirming that σ has found all submissions in the batch to be non-equivocated, *except* for those in E_σ. In the good case, every client is correct and E_σ is empty. Having produced c_σ, σ moves on to build a set Q_σ containing a *proof of equivocation* for every element in E_σ. Let us assume that σ previously received from some χ_k a payload p'_k that conflicts with p_k. σ must have received p'_k as part of some witnessed batch. Let r'_k identify the root of p'_k's batch, let w'_k identify r'_k's witness, let q'_k be (i_k, p'_k)'s proof of inclusion in r'_k. By exhibiting (r'_k, w'_k, p'_k), σ can prove to β that χ_k equivocated: p_k conflicts with p'_k, and (i_k, p'_k) is provably part of a batch whose integrity was witnessed by at least one correct server. Furthermore, because correct clients never equivocate, (r'_k, w'_k, p'_k) is sufficient to convince β that χ_k is Byzantine. For each i_j in E_σ, σ collects in Q_σ a proof of equivocation $\left(r'_j, w'_j, p'_j\right)$. Finally, σ sends a `CommitShard` message back to β (fig. 1, step 9). The `CommitShard` message contains c_σ, E_σ and Q_σ. Upon receiving σ's `CommitShard` message, σ verifies c_σ against r and E_σ, then checks all proofs in Q_σ. Having collected valid `CommitShard` messages from a quorum of servers $\sigma_1, \ldots, \sigma_{2f+1}$, β aggregates all commit shards into a *commit certificate* c. We underline that each σ_j signed the same root r, but a potentially different set of exceptions E_{σ_j}. Let E denote the union of $E_{\sigma_1}, \ldots, E_{\sigma_{2f+1}}$. We call E the batch's *exclusion* set. Because a proof of equivocation cannot be produced against a correct client, β knows that all clients identified by E are necessarily Byzantine. In particular, because χ is correct, $(i = i_n)$ is guaranteed to not be in E.

Committing the batch. Having collected a commit certificate c for the batch, β sends c to all servers by means of a `Commit` message (fig. 1, step 10). Upon receiving the `Commit` message, σ verifies c, computes the exclusion set E, then delivers every payload p_j whose id i_j is not in E. Recalling that c is assembled from a quorum of commit shards, at least $f + 1$ correct servers contributed to c. This means that, if some i_k is not in E, then at least $f + 1$ correct servers found p_k not to be equivocated. As in most BRB implementations [9], this guarantees that no two commit certificates can be gathered for equivocating payloads: Draft's consistency is upheld.

The role of equivocation proofs. As the reader might have noticed, β does not attach any proof of equivocation to its `Commit` message. Having received β's commit certificate c, σ trusts β's exclusion set E, ignoring every payload whose id is in E. This is not because σ can trust β to uphold validity. On the contrary, σ has *no way* to determine that β is not maliciously excluding the payload of a correct client. Indeed, even if σ were to verify a proof of exclusion for every element in E, a malicious β could still censor a correct client simply by ignoring its `Submit` message in the first place. Equivocation proofs are fundamental to Draft's validity not because they *force* malicious brokers to uphold validity, but because they *enable* correct brokers to do the same. Thanks to equivocation proofs, a malicious server cannot trick a correct broker into excluding the payload of a correct client. This is enough to guarantee validity. As we discuss below, χ successively submits p to all brokers until it receives a certificate attesting that p was delivered by at least one correct server. Because we assume at least one broker to be correct, χ is eventually guaranteed to succeed.

Notifying the clients. Having delivered every payload whose id is not in the exclusion set E, σ produces a *completion shard* for the batch, i.e., a multi-signature z_σ for $[\texttt{Completion}, r, E]$, effectively affirming that σ has delivered all submissions in the batch whose id is not in E. σ sends z_σ to β by means of a `CompletionShard` message (fig. 1, step 11). Upon receiving $f + 1$ valid `CompletionShard` messages, β assembles all completion shards into a *completion certificate* z. Finally, β sends a `Completion` message to $\chi_1, \ldots, \chi_b$ (fig. 1, step 12). The `Completion` message contains z and E. Upon receiving the `Completion` message, χ verifies z against E, then checks that i is not in E. Because at least one correct server contributed a completion shard to z, at least one correct process delivered all payloads that E did not exclude, including p. Having succeeded in broadcasting p, χ does not need to engage further, and can stop successively submitting p to all brokers.

No one is left behind. As we discussed above, upon receiving the commit certificate c, σ delivers every payload in the batch whose id is not in the exclusion set E. Having gotten at least one correct server to deliver the batch, β is free to disengage, and moves on to assembling and brokering its next batch. In a moment of asynchrony, however, all communications between β and $\tilde{\sigma}$ might be arbitrarily delayed. This means that $\tilde{\sigma}$ has no way of telling whether or not it will eventually receive batch and commit certificate: a malicious β might have deliberately left $\tilde{\sigma}$ out of the protocol. Server-to-server communication is thus required to guarantee totality. Having delivered the batch, σ waits for an interval of time long enough for all correct servers to deliver batch and commit certificate, *should the network be synchronous and* β *correct.* σ then sends to all servers an `OfferTotality` message (fig. 1, step 13). The `OfferTotality` message contains the batch's root r, and the exclusion set E. In the good case, upon receiving σ's `OfferTotality` message, every server has delivered the batch and ignores the offer. This, however, is not the case for slow $\hat{\sigma}$, which replies to

σ with an `AcceptTotality` message (fig. 1, step 14). Upon receiving $\tilde{\sigma}$'s `AcceptTotality` message, σ sends back to $\tilde{\sigma}$ a `Totality` message (fig. 1, step 15). The `Totality` message contains all submissions $(i_1, p_1), \ldots, (i_b, p_b)$, id assignments for $i_1, \ldots, i_b$, and the commit certificate c. Upon delivering σ's `Totality` message, $\tilde{\sigma}$ computes r from $(i_1, p_1), \ldots, (i_b, p_b)$, checks c against r, computes E from c, and delivers every payload p_j whose id i_j is not in E. This guarantees totality and concludes the protocol.

4.2 Complexity

Directory density. As we introduced in Section 4.1, Draft uses ids assigned by its underlying Directory (DIR) abstraction to identify payload senders. A DIR-assigned id is composed of two parts: a *domain* and an *index*. Domains form a finite set $\mathbb{D}$ whose size does not increase with the number of clients, indices are natural numbers. Along with safety (e.g., no two processes have the same id) and liveness (e.g., every correct client that requests an id eventually obtains an id), DIR guarantees *density*: the index part of any id is always smaller than the total number of clients c (i.e., each id index is between 0 and $(c-1)$). Intuitively, this echoes the (stronger) density guarantee provided by Oracle-CSB, the oracle-based implementation of CSB we introduced in Section 1 to bound Draft's performance. In Oracle-CSB, the oracle organizes all clients in a list, effectively labeling each client with an integer between 0 and $(c-1)$. In a setting where consensus cannot be achieved, agreeing on a totally-ordered list of clients is famously impossible: a consensus-less DIR implementation cannot assign ids if $|\mathbb{D}| = 1$. However, DIR can be implemented without consensus if servers are used as domains ($\mathbb{D} = \Sigma$). In our DIR implementation Dibs, each server maintains an independent list of public keys. In order to obtain an id, a client χ has each server add its public key to its list, then selects a server σ to be its *assigner*. In doing so, χ obtains an id (σ, n), where $n \in 0..(c-1)$ is χ's position in σ's log. In summary, a consensus-less implementation of DIR still guarantees that indices will be smaller than c, at the cost of a non-trivial domain component for each id. This inflates the size of each individual id by $\lceil \log_2 (|\mathbb{D}|) \rceil$ bits.

Batching ids. While DIR-assigned ids come with a non-trivial domain component, the size overhead due to domains vanishes when infinitely many ids are organized into a batch. This is because domains are constant in the number of clients. Intuitively, as infinitely many ids are batched together, repeated domains become compressible. When building a batch, a Draft broker represents the set I of sender ids not as a list, but as a map $\tilde{i}$. To each domain, $\tilde{i}$ associates all ids in I under that domain (n is in $\tilde{i}[d]$ iff (d, n) is in I). Because $\tilde{i}$'s keys are fixed, as the size of I goes to infinity, the bits required to represent $\tilde{i}$'s keys are completely amortized by those required to represent $\tilde{i}$'s values. At the batching limit, the cost of representing each id in $\tilde{i}$ converges to that of representing its index only, $\lceil \log_2 (c) \rceil$.

Protocol cost. At the batching limit we assume a good-case execution: links are synchronous, all processes are correct, and the set of brokers contains only one element. We additionally assume that infinitely many clients broadcast concurrently. Finally, we assume all servers to already know all broadcasting clients. Let β denote the only broker. As all broadcasting clients submit their payloads to β within a suitably narrow time window, β organizes all submissions into a single batch with root r. Because links are synchronous and all clients are correct, every broadcasting client submits its multi-signature for r in time. Having removed all individual signatures from the batch, β is left with a single, aggregated multi-signature m and an empty straggler set S. β compresses the sender ids and disseminates the batch to

all servers. As m and S are constant-sized, the amortized cost for a server to receive each payload p is $(\lceil \log_2 (c) \rceil + |p|)$ bits. As m authenticates the entire batch, a server authenticates each payload at an amortized cost of 0 signature verifications. The remainder of the protocol unfolds as a sequence of constant-sized messages: because all broadcasting clients are known to all servers, no server requests any id assignment; witnesses are always constant-sized; and because all processes are correct, no client equivocates and all exception sets are empty. Finally, again by the synchrony of links, all offers of totality are ignored. In summary, at the batching limit a server delivers a payload at an amortized cost of 0 signature verifications and $(\lceil \log_2 (c) \rceil + |p|)$ bits exchanged.

Latency. As depicted in Figure 1, the latency of Draft is 10 message delays in the synchronous case (fast servers deliver upon receiving the broker's Commit message), and at most 13 message delays in the asynchronous case (slow servers deliver upon receiving other servers' Totality messages). By comparison, the latency of the optimistic reliable broadcast algorithm by Cachin *et al.* [13] is respectively 4 message delays (synchronous case) and 6 message delays (asynchronous case). Effectively, Draft trades oracular efficiency for a constant latency overhead.

Worst-case complexity. In the worst case, a Draft server delivers a b-bits payload by exchanging $O((\log(c) + b)kn)$ bits, where c, k and n respectively denote the number of clients, brokers and servers. In brief, the same id, payload and signature is included by each broker in a different batch (hence the k term) and propagated in an all-to-all fashion (carried by `Totality` messages) across correct servers (hence the n term). By comparison, the worst-case communication complexity of Cachin *et al.*'s optimistic reliable broadcast is $O(ln)$ per server, where l is the length of the broadcast payload. A direct batched generalization of the same algorithm, however, would raise the worst-case communication to $O(ln^2)$ per server, similar to that of Draft when $n \sim k$. Both batched Bracha and Draft can be optimized by polynomial encoding, reducing their per-server worst-case complexity to $O(ln)$ and $O((\log(c) + b)k)$ respectively. Doing so for Draft, however, is beyond the scope of this paper.

5 Conclusions

Our contributions. In this paper we study Client-Server Byzantine Reliable Broadcast (CSB), a multi-shot variant of Byzantine Reliable Broadcast (BRB) whose interface is split between broadcasting clients and delivering servers. We introduce Oracle-CSB, a toy implementation of CSB that relies on a single, infallible oracle to uphold all CSB properties. Unless clients can be assumed to broadcast at a non-uniform rate, Oracle-CSB's signature and communication complexities are optimal: in Oracle-CSB, a server delivers a payload p by performing 0 signature verifications, and exchanging $(\lceil \log_2 (c) \rceil + |p|)$ bits, where c is the number of clients. We present Draft, our implementation of CSB. Draft upholds all CSB properties under classical BRB assumptions (notably asynchronous links and less than a third of faulty servers). When links are synchronous and all processes are correct, however, and at the limit of infinite concurrently broadcasting clients, Draft's signature and communication complexities match those of Oracle CSB.

Future work. We hope to extend Draft to allow multiple messages by the same client in the same batch. We envision that this could be achieved by using other types of cryptographic accumulators or variants of Merkle trees, such as Merkle-Patricia trees [25]. It would also be interesting to see if the worst-case performance of Draft could be improved, e.g. by using error correction codes (ECC) or erasure codes, without significantly affecting its good-case performance. Lastly, we hope to use Draft's keys ideas to implement a total-order broadcast primitive, improving the scalability of existing SMR implementations.

References

1. Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-Case Latency of Byzantine Broadcast: A Complete Categorization. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 331–341, New York, NY, USA, 2021. Association for Computing Machinery.
2. Nicolas Alhaddad, Sisi Duan, Mayank Varia, and Haibin Zhang. Succinct erasure coding proof systems. *Cryptology ePrint Archive*, 2021.
3. Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.
4. Alex Auvolat, Davide Frey, Michel Raynal, and François Taïani. Byzantine-tolerant causal broadcast. *Theoretical Computer Science*, 885:55–68, 2021.
5. Paulo S. L. M. Barreto, Hae Y. Kim, Ben Lynn, and Michael Scott. Efficient algorithms for pairing-based cryptosystems. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, pages 354–369, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.
6. Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 52–61, 1993.
7. Joseph T.A. Birman K.P. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.
8. Gabriel Bracha. An asynchronous [(n-1)/3]-resilient consensus protocol. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 154–162, 1984.
9. Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.
10. Gabriel Bracha and Sam Toueg. Asynchronous Consensus and Broadcast Protocols. *JACM*, 32(4), 1985.
11. Christian Cachin. State machine replication with byzantine faults. In *Replication*, pages 169–184. Springer, 2010.
12. Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming.* Springer Science & Business Media, 2011.
13. Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*, pages 524–541. Springer, 2001.
14. Christian Cachin and Jonathan A. Poritz. Secure Intrusion-tolerant Replication on the Internet. In *DSN*, 2002.
15. Christian Cachin and Jonathan A Poritz. Secure intrusion-tolerant replication on the internet. In *Proceedings International Conference on Dependable Systems and Networks*, pages 167–176. IEEE, 2002.

16 Christian Cachin and Stefano Tessaro. Asynchronous Verifiable Information Dispersal. In *Proceedings of the 24th Symposium on Reliable Distributed Systems – SRDS 2005*, pages 191–202, October 2005.

17 Martina Camaioni, Rachid Guerraoui, Matteo Monti, and Manuel Vidigueira. Oracular Byzantine Reliable Broadcast (Extended Version), 2022.

18 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

19 Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 26–38, 2020. doi:10.1109/DSN48063.2020.00023.

20 Thomas Cover and Joy Thomas. *Elements of Information Theory, Second Edition.* John Wiley & Sons, 2005.

21 Tyler Crain, Christopher Natoli, and Vincent Gramoli. Evaluating the Red Belly Blockchain. *CoRR*, 2018.

22 Tyler Crain, Christopher Natoli, and Vincent Gramoli. Red belly: A secure, fair and scalable open blockchain. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 466–483. IEEE, 2021.

23 George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 34–50, 2022.

24 Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous data dissemination and its applications. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2721, 2021.

25 Haitz Sáez de Ocáriz Borde. An overview of trees in blockchain technology: Merkle trees and merkle patricia tries, 2022.

26 Xavier Défago, André Schiper, and Péter Urbán. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Comput. Surv.*, 36(4):372–421, December 2004.

27 Danny Dolev and Rüdiger Reischuk. Bounds on Information Exchange for Byzantine Agreement. *J. ACM*, 32(1):191–204, January 1985.

28 Sisi Duan, Michael K. Reiter, and Haibin Zhang. BEAT: Asynchronous BFT Made Practical. In *CCS*, 2018.

29 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos Seredinschi. The Consensus Number of a Cryptocurrency. In *PODC*, 2019.

30 James Hendricks, Gregory R Ganger, and Michael K Reiter. Verifying distributed erasure-coded data. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 139–146, 2007.

31 Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 165–175, 2021.

32 Klaus Kursawe and Victor Shoup. Optimistic asynchronous atomic broadcast. In *International Colloquium on Automata, Languages, and Programming*, pages 204–215. Springer, 2005.

33 Petr Kuznetsov, Yvonne-Anne Pignolet, Pavel Ponomarev, and Andrei Tonkikh. Permissionless and asynchronous asset transfer. In *35th International Symposium on Distributed Computing (DISC 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

34 D. Malkhi, M. Merritt, and O. Rodeh. Secure reliable multicast protocols in a wan. In *Proceedings of 17th International Conference on Distributed Computing Systems*, pages 87–94, 1997.

35 Dahlia Malkhi and Michael Reiter. A High-Throughput Secure Reliable Multicast Protocol. *Journal of Computer Security*, 5:113–127, 1996.

36 Dahlia Malkhi and Michael Reiter. A high-throughput secure reliable multicast protocol. *Journal of Computer Security*, 5(2):113–127, 1997.

37 Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

38 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 31–42, 2016.

39 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

40 Kartik Nayak, Ling Ren, Elaine Shi, Nitin H Vaidya, and Zhuolun Xiang. Improved extension protocols for byzantine broadcast and agreement. In *34th International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

41 James S Plank and Lihao Xu. Optimizing cauchy reed-solomon codes for fault-tolerant network storage applications. In *Fifth IEEE International Symposium on Network Computing and Applications (NCA'06)*, pages 173–180. IEEE, 2006.

42 HariGovind V Ramasamy and Christian Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *International Conference On Principles Of Distributed Systems*, pages 88–102. Springer, 2005.

43 Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

44 Michael K. Reiter and Kenneth P. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3), 1994.

45 Nuno Santos and André Schiper. Optimizing Paxos with batching and pipelining. *Theoretical Computer Science*, 496:170–183, 2013. Distributed Computing and Networking (ICDCN 2012).

46 Victor Shoup. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 207–220. Springer, 2000.

47 Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. State machine replication scalability made simple. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 17–33, 2022.

48 Andrew Chi-Chih Yao. Some Complexity Questions Related to Distributive Computing. In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, STOC '79, pages 209–213, New York, NY, USA, 1979. Association for Computing Machinery.

Byzantine Consensus Is $\Theta(n^2)$

The Dolev-Reischuk Bound Is Tight Even in Partial Synchrony!

Pierre Civit
Sorbonne University, Paris, France

Muhammad Ayaz Dzulfikar
NUS Singapore, Singapore

Seth Gilbert
NUS Singapore, Singapore

Vincent Gramoli
University of Sydney, Australia
Redbelly Network, Sydney, Australia

Rachid Guerraoui
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Jovan Komatovic
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Manuel Vidigueira
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

The Dolev-Reischuk bound says that any deterministic Byzantine consensus protocol has (at least) quadratic communication complexity in the worst case. While it has been shown that the bound is tight in synchronous environments, it is still unknown whether a consensus protocol with quadratic communication complexity can be obtained in partial synchrony. Until now, the most efficient known solutions for Byzantine consensus in partially synchronous settings had cubic communication complexity (e.g., HotStuff, binary DBFT).

This paper closes the existing gap by introducing SQuad, a partially synchronous Byzantine consensus protocol with quadratic worst-case communication complexity. In addition, SQuad is optimally-resilient and achieves linear worst-case latency complexity. The key technical contribution underlying SQuad lies in the way we solve *view synchronization*, the problem of bringing all correct processes to the same view with a correct leader for sufficiently long. Concretely, we present RareSync, a view synchronization protocol with quadratic communication complexity and linear latency complexity, which we utilize in order to obtain SQuad.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Optimal Byzantine consensus, Communication complexity, Latency complexity

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.14

Related Version The extended version of this paper, which includes detailed proofs, is available online.
Full Version: `https://arxiv.org/abs/2208.09262` [19]

Funding This work is supported in part by the ARC Future Fellowship funding scheme (#180100496).
Seth Gilbert: Supported in part by Singapore MOE grant MOE2018-T2-1-160.

Acknowledgements The authors would like to thank Gregory Chockler and Alexey Gotsman for helpful conversations.

© Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

Byzantine consensus [38] is a fundamental distributed computing problem. In recent years, it has become the target of widespread attention due to the advent of blockchain [22, 4, 31] and decentralized cloud computing [41], where it acts as a key primitive. The demand of these contexts for high performance has given a new impetus to research towards Byzantine consensus with optimal communication guarantees.

Intuitively, Byzantine consensus enables processes to agree on a common value despite Byzantine failures. Formally, each process is either correct or faulty; correct processes follow a prescribed protocol, whereas faulty processes (up to $f > 0$) can arbitrarily deviate from it. Each correct process *proposes* a value, and should eventually *decide* a value. The following properties are guaranteed:

- *Validity:* If all correct processes propose the same value, then only that value can be decided by a correct process.
- *Agreement:* No two correct processes decide different values.
- *Termination:* All correct processes eventually decide.

The celebrated Dolev-Reischuk bound [25] says that any deterministic solution of the Byzantine consensus problem requires correct processes to exchange (at least) a quadratic number of bits of information. It has been shown that the bound is tight in synchronous environments [10, 46]. However, for the partially synchronous environments [26] in which the network becomes synchronous only after some unknown Global Stabilization Time (GST), no Byzantine consensus protocol achieving quadratic communication complexity is known.[1] Therefore, the question remains whether a partially synchronous Byzantine consensus with quadratic communication complexity exists [20]. Until now, the most efficient known solutions in partially synchronous environments had cubic communication complexity (e.g., HotStuff [56], binary DBFT [22]).

We close the gap by introducing SQuad, a partially synchronous Byzantine consensus protocol with quadratic worst-case communication complexity, matching the Dolev-Reischuk [25] bound. In addition, SQuad is optimally-resilient and achieves optimal linear worst-case latency.

Partially synchronous "leader-based" Byzantine consensus. Partially synchronous "leader-based" consensus protocols [56, 55, 15, 13] operate in *views*, each with a designated leader whose responsibility is to drive the system towards a decision. If a process does not decide in a view, the process moves to the next view with a different leader and tries again. Once all correct processes overlap in the same view with a correct leader for sufficiently long, a decision is reached. Sadly, ensuring such an overlap is non-trivial; for example, processes can start executing the protocol at different times or their local clocks may drift before GST, thus placing them in views which are arbitrarily far apart.

Typically, these protocols contain two independent modules:

1. View core: The core of the protocol, responsible for executing the protocol logic of each view.
2. View synchronizer: Auxiliary to the view core, responsible for "moving" processes to new views with the goal of ensuring a sufficiently long overlap to allow the view core to decide.

[1] No deterministic protocol solves Byzantine consensus in a completely asynchronous environment [27].

Immediately after GST, the view synchronizer brings all correct processes together to the view of the most advanced correct process and keeps them in that view for sufficiently long. At this point, if the leader of the view is correct, the processes decide. Otherwise, they "synchronously" transit to the next view with a different leader and try again. In summary, the communication complexity of such protocols can be approximated by $n \cdot C + S$, where:

- C denotes the maximum number of bits a correct process sends while executing its view core during $[GST, t_d]$, where t_d is the first time by which all correct processes have decided, and
- S denotes the communication complexity of the view synchronizer during $[GST, t_d]$.

Since the adversary can corrupt up to f processes, correct processes must transit through at least $f + 1$ views after GST, in the worst case, before reaching a correct leader. In fact, PBFT [15] and HotStuff [56] show that passing through $f + 1$ views is sufficient to reach a correct leader. Furthermore, HotStuff employs the "leader-to-all, all-to-leader" communication pattern in each view. As (1) each process is the leader of at most one view during $[GST, t_d]$, and (2) a process sends $O(n)$ bits in a view if it is the leader of the view, and $O(1)$ bits otherwise, HotStuff achieves $C = 1 \cdot O(n) + f \cdot O(1) = O(n)$. Unfortunately, $S = (f + 1) \cdot O(n^2) = O(n^3)$ in HotStuff due to "all-to-all" communication exploited by its view synchronizer in *every* view.[2] Thus, $S = O(n^3)$ dominates the communication complexity of HotStuff, preventing it from matching the Dolev-Reischuk bound. If we could design a consensus algorithm for which $S = O(n^2)$ while preserving $C = O(n)$, we would obtain a Byzantine consensus protocol with optimal communication complexity. The question is if a view synchronizer achieving $S = O(n^2)$ in partial synchrony exists.

Warm-up: View synchronization in complete synchrony. Solving the synchronization problem in a completely synchronous environment is not hard. As all processes start executing the protocol at the same time and their local clocks do not drift, the desired overlap can be achieved without any communication: processes stay in each view for the fixed, overlap-required time. However, this simple method *cannot* be used in a partially synchronous setting as it is neither guaranteed that all processes start at the same time nor that their local clocks do not drift (before GST). Still, the observation that, if the system is completely synchronous, processes are not required to communicate in order to synchronize plays a crucial role in developing our view synchronizer which achieves quadratic communication complexity in partially synchronous environments.

RareSync. The main technical contribution of this work is RareSync, a partially synchronous view synchronizer that achieves synchronization within $O(f)$ time after GST, and has $O(n^2)$ worst-case communication complexity. In a nutshell, RareSync adapts the "no-communication" technique of synchronous view synchronizers to partially synchronous environments.

Namely, RareSync groups views into *epochs*; each epoch contains $f + 1$ sequential views. Instead of performing "all-to-all" communication in each view (like the "traditional" view synchronizers [55]), RareSync performs a *single* "all-to-all" communication step per epoch. Specifically, *only* at the end of each epoch do all correct processes communicate to enable further progress. Once a process has entered an epoch, the process relies *solely* on its local clock (without any communication) to move forward to the next view within the epoch.

[2] While HotStuff [56] does not explicitly state how the view synchronization is achieved, we have that $S = O(n^3)$ in Diem BFT [55], which is a mature implementation of the HotStuff protocol.

Let us give a (rough) explanation of how RareSync ensures synchronization. Let E be the smallest epoch entered by *all* correct processes at or after GST; let the first correct process enter E at time $t_E \geq GST$. Due to (1) the "all-to-all" communication step performed at the end of the previous epoch $E-1$, and (2) the fact that message delays are bounded by a known constant δ after GST, all correct processes enter E by time $t_E + \delta$. Hence, from the epoch E onward, processes do not need to communicate in order to synchronize: it is sufficient for processes to stay in each view for $\delta + \Delta$ time to achieve Δ-time overlap. In brief, RareSync uses communication to synchronize processes, while relying on local timeouts (and not communication!) to keep them synchronized.

SQuad. The second contribution of our work is SQuad, an optimally-resilient partially synchronous Byzantine consensus protocol with (1) $O(n^2)$ worst-case communication complexity, and (2) $O(f)$ worst-case latency complexity. The view core module of SQuad is the same as that of HotStuff; as its view synchronizer, SQuad uses RareSync. The combination of the HotStuff's view core and RareSync ensures that $C = O(n)$ and $S = O(n^2)$. By the aforementioned complexity formula, SQuad achieves $n \cdot O(n) + O(n^2) = O(n^2)$ communication complexity. SQuad's linear latency is a direct consequence of RareSync's ability to synchronize processes within $O(f)$ time after GST.

Roadmap. We discuss related work in §2. In §3, we define the system model. We introduce RareSync in §4. In §5, we present SQuad. We conclude the paper in §6.

2 Related Work

In this section, we discuss existing results in two related contexts: synchronous networks and randomized algorithms. In addition, we discuss some precursor (and concurrent) results to our own.

Synchronous networks. The first natural question is whether we can achieve synchronous Byzantine agreement with optimal latency and optimal communication complexity. Momose and Ren answer that question in the affirmative, giving a synchronous Byzantine agreement protocol with optimal $n/2$ resiliency, optimal $O(n^2)$ worst-case communication complexity and optimal $O(f)$ worst-case latency [46]. Optimality follows from two lower bounds: Dolev and Reischuk show that any Byzantine consensus protocol has an execution with quadratic communication complexity [25]; Dolev and Strong show that any synchronous Byzantine consensus protocol has an execution with $f+1$ rounds [23]. Various other works have tackled the problem of minimizing the latency of Byzantine consensus [2, 42, 45].

Randomization. A classical approach to circumvent the FLP impossibility [27] is using randomization [9], where termination is not ensured deterministically. Exciting recent results by Abraham *et al.* [5] and Lu *et al.* [43] give fully asynchronous randomized Byzantine consensus with optimal $n/3$ resiliency, optimal $O(n^2)$ expected communication complexity and optimal $O(1)$ expected latency complexity. Spiegelman [53] took a neat *hybrid* approach that achieved optimal results for both synchrony and randomized asynchrony simultaneously: if the network is synchronous, his algorithm yields optimal (deterministic) synchronous complexity; if the network is asynchronous, it falls back on a randomized algorithm and achieves optimal randomized complexity.

Recently, it has been shown that even randomized Byzantine agreement requires $\Omega(n^2)$ expected communication complexity, at least for achieving guaranteed safety against an *adaptive adversary* in an asynchronous setting or against a *strongly rushing adaptive adversary* in a synchronous setting [1, 6]. (See the papers for details.) Amazingly, it is possible to break the $O(n^2)$ barrier by accepting a non-zero (but $o(1)$) probability of disagreement [18, 21, 35].

Authentication. Most of the results above are *authenticated*: they assume a trusted setup phase[3] wherein devices establish and exchange cryptographic keys; this allows for messages to be signed in a way that proves who sent them. Recently, many of the communication-efficient agreement protocols (such as [5, 43]) rely on *threshold signatures* (such as [40]). The Dolev-Reischuk [25] lower bound shows that quadratic communication is needed even in such a case (as it looks at the message complexity of authenticated agreement).

Among deterministic, non-authenticated Byzantine agreement protocols, DBFT [22] achieves $O(n^3)$ communication complexity. For randomized non-authenticated Byzantine agreement protocols, Mostefaoui *et al.* [47] achieve $O(n^2)$ communication complexity – but they assume a perfect common coin, for which efficient implementations may also require signatures.

We note that it is possible to (1) work towards an authenticated setting from a non-authenticated one by rolling out a public key infrastructure (PKI) [11, 7, 29], (2) set up a threshold scheme [3] without a *trusted dealer*, and (3) asynchronously emulate a perfect common coin [14] used by randomized Byzantine consensus protocols [51, 47, 5, 43].

Other related work. In this paper, we focus on the partially synchronous setting [26], where the question of optimal communication complexity of Byzantine agreement has remained open. The question can be addressed precisely with the help of rigorous frameworks [28, 32, 33] that were developed to express partially synchronous protocols using a round-based paradigm. More specifically, state-of-the-art partially synchronous BFT protocols [55, 13, 56, 30] have been developed within a view-based paradigm with a rotating leader, e.g., the seminal PBFT protocol [15]. While many approaches improve the complexity for some optimistic scenarios [44, 52, 36, 37, 50], none of them were able to reach the quadratic worst-case Dolev-Reischuk bound.

The problem of view synchronization was defined in [48]. An existing implementation of this abstraction [30] was based on Bracha's double-echo reliable broadcast at each view, inducing a cubic communication complexity in total. This communication complexity has been reduced for some optimistic scenarios [48] and in terms of *expected* complexity [49]. The problem has been formalized more precisely in [12] to facilitate formal verification of PBFT-like protocols.

It might be worthwhile highlighting some connections between the view synchronization abstraction and the leader election abstraction Ω [16, 17], capturing the weakest failure detection information needed to solve consensus (and extended to the Byzantine context in [34]). Leaderless partially synchronous Byzantine consensus protocols have also been proposed [8], somehow indicating that the notion of a leader is not necessary in the mechanisms of a consensus protocol, even if Ω is the weakest failure detector needed to solve the problem. Clock synchronization [24, 54] and view synchronization are orthogonal problems.

[3] A trusted setup phase is notably different from randomized algorithms where randomization is used throughout.

Concurrent research. We have recently discovered concurrent and independent research by Lewis-Pye [39]. Lewis-Pye appears to have discovered a similar approach to the one that we present in this paper, giving an algorithm for state machine replication in a partially synchronous model with quadratic message complexity. As in this paper, Lewis-Pye makes the key observation that we do not need to synchronize in every view; views can be grouped together, with synchronization occurring only once every fixed number of views. This yields essentially the same algorithmic approach. Lewis-Pye focuses on state machine replication, instead of Byzantine agreement (though state machine replication is implemented via repeated Byzantine agreement). The other useful property of his algorithm is *optimistic responsiveness*, which applies to the multi-shot case and ensures that, in good portions of the executions, decisions happen as quickly as possible. We encourage the reader to look at [39] for a different presentation of a similar approach.

3 System Model

Processes. We consider a static set $\{P_1, P_2, ..., P_n\}$ of $n = 3f + 1$ processes out of which at most f can be Byzantine, i.e., can behave arbitrarily. If a process is Byzantine, the process is *faulty*; otherwise, the process is *correct*. Processes communicate by exchanging messages over an authenticated point-to-point network. The communication network is *reliable:* if a correct process sends a message to a correct process, the message is eventually received. We assume that processes have local hardware clocks. Furthermore, we assume that local steps of processes take zero time, as the time needed for local computation is negligible compared to message delays. Finally, we assume that no process can take infinitely many steps in finite time.

Partial synchrony. We consider the partially synchronous model introduced in [26]. For every execution, there exists a Global Stabilization Time (GST) and a positive duration δ such that message delays are bounded by δ after GST. Furthermore, GST is not known to processes, whereas δ is known to processes. We assume that all correct processes start executing their protocol by GST. The hardware clocks of processes may drift arbitrarily before GST, but do not drift thereafter.

Cryptographic primitives. We assume a (k, n)-threshold signature scheme [40], where $k = 2f+1 = n-f$. In this scheme, each process holds a distinct private key and there is a single public key. Each process P_i can use its private key to produce a partial signature of a message m by invoking $\mathit{ShareSign}_i(m)$. A partial signature *tsignature* of a message m produced by a process P_i can be verified by $\mathit{ShareVerify}_i(m, \mathit{tsignature})$. Finally, set $S = \{\mathit{tsignature}_i\}$ of partial signatures, where $|S| = k$ and, for each $\mathit{tsignature}_i \in S$, $\mathit{tsignature}_i = \mathit{ShareSign}_i(m)$, can be combined into a *single* (threshold) signature by invoking $\mathit{Combine}(S)$; a combined signature *tcombined* of message m can be verified by $\mathit{CombinedVerify}(m, \mathit{tcombined})$. Where appropriate, invocations of $\mathit{ShareVerify}(\cdot)$ and $\mathit{CombinedVerify}(\cdot)$ are implicit in our descriptions of protocols. P_Signature and T_Signature denote a partial signature and a (combined) threshold signature, respectively.

Complexity of Byzantine consensus. Let Consensus be a partially synchronous Byzantine consensus protocol and let $\mathcal{E}(\text{Consensus})$ denote the set of all possible executions. Let $\alpha \in \mathcal{E}(\text{Consensus})$ be an execution and $t_d(\alpha)$ be the first time by which all correct processes have decided in α.

A *word* contains a constant number of signatures and values. Each message contains at least a single word. We define the communication complexity of α as the number of words sent in messages by all correct processes during the time period $[GST, t_d(\alpha)]$; if $GST > t_d(\alpha)$, the communication complexity of α is 0. The latency complexity of α is $\max(0, t_d(\alpha) - GST)$.

The *communication complexity* of Consensus is defined as

$$\max_{\alpha \in \mathcal{E}(\mathsf{Consensus})} \Big\{ \text{communication complexity of } \alpha \Big\}.$$

Similarly, the *latency complexity* of Consensus is defined as

$$\max_{\alpha \in \mathcal{E}(\mathsf{Consensus})} \Big\{ \text{latency complexity of } \alpha \Big\}.$$

We underline that the number of words sent by correct processes before GST is unbounded in any partially synchronous Byzantine consensus protocol [53]. Moreover, not a single correct process is guaranteed to decide before GST in any partially synchronous Byzantine consensus protocol [27]; that is why the latency complexity of such protocols is measured from GST.

4 RareSync

This section presents RareSync, a partially synchronous view synchronizer that achieves synchronization within $O(f)$ time after GST, and has $O(n^2)$ worst-case communication complexity. First, we define the problem of view synchronization (§4.1). Then, we describe RareSync, and present its pseudocode (§4.2). Finally, we reason about RareSync's correctness and complexity (§4.3).

4.1 Problem Definition

View synchronization is defined as the problem of bringing all correct processes to the same view with a correct leader for sufficiently long [12, 49, 48]. More precisely, let $\mathsf{View} = \{1, 2, ...\}$ denote the set of views. For each view $v \in \mathsf{View}$, we define $\mathsf{leader}(v)$ to be a process that is the *leader* of view v. The view synchronization problem is associated with a predefined time $\Delta > 0$, which denotes the desired duration during which processes must be in the same view with a correct leader in order to synchronize. View synchronization provides the following interface:

- **Indication** advance(View v): The process advances to a view v.

We say that a correct process *enters* a view v at time t if and only if the advance(v) indication occurs at time t. Moreover, a correct process *is in view* v between the time t (including t) at which the advance(v) indication occurs and the time t' (excluding t') at which the next advance($v' \neq v$) indication occurs. If an advance($v' \neq v$) indication never occurs, the process remains in the view v from time t onward.

Next, we define a *synchronization time* as a time at which all correct processes are in the same view with a correct leader for (at least) Δ time.

▶ **Definition 1** (Synchronization time)**.** *Time t_s is a* synchronization time *if (1) all correct processes are in the same view v from time t_s to (at least) time $t_s + \Delta$, and (2)* leader(v) *is correct.*

View synchronization ensures the *eventual synchronization* property which states that there exists a synchronization time at or after GST.

Complexity of view synchronization. Let Synchronizer be a partially synchronous view synchronizer and let $\mathcal{E}(\textsf{Synchronizer})$ denote the set of all possible executions. Let $\alpha \in \mathcal{E}(\textsf{Synchronizer})$ be an execution and $t_s(\alpha)$ be the first synchronization time at or after GST in α ($t_s(\alpha) \geq GST$). We define the communication complexity of α as the number of words sent in messages by all correct processes during the time period $[GST, t_s(\alpha) + \Delta]$. The latency complexity of α is $t_s(\alpha) + \Delta - GST$.

The *communication complexity* of Synchronizer is defined as

$$\max_{\alpha \in \mathcal{E}(\textsf{Synchronizer})} \Big\{ \text{communication complexity of } \alpha \Big\}.$$

Similarly, the *latency complexity* of Synchronizer is defined as

$$\max_{\alpha \in \mathcal{E}(\textsf{Synchronizer})} \Big\{ \text{latency complexity of } \alpha \Big\}.$$

4.2 Protocol

This subsection details RareSync (Algorithm 2). In essence, RareSync achieves $O(n^2)$ communication complexity and $O(f)$ latency complexity by exploiting "all-to-all" communication only once per $f+1$ views.

Intuition. We group views into *epochs*, where each epoch contains $f+1$ sequential views; Epoch $= \{1, 2, ...\}$ denotes the set of epochs. Processes move through an epoch solely by means of local timeouts (without any communication). However, at the end of each epoch, processes engage in an "all-to-all" communication step to obtain permission to move onto the next epoch: (1) Once a correct process has completed an epoch, it broadcasts a message informing other processes of its completion; (2) Upon receiving $2f+1$ of such messages, a correct process enters the future epoch. Note that (2) applies to *all* processes, including those in arbitrarily "old" epochs. Overall, this "all-to-all" communication step is the *only* communication processes perform within a single epoch, implying that per-process communication complexity in each epoch is $O(n)$. Figure 1 illustrates the main idea behind RareSync.

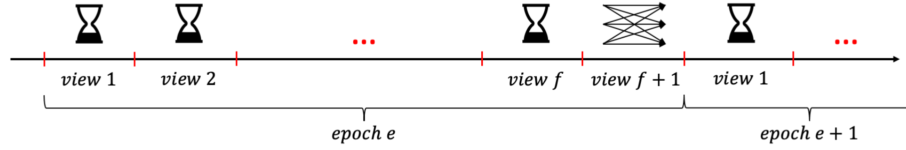


Figure 1 Intuition behind RareSync: Processes communicate only in the last view of an epoch; before the last view, they rely solely on local timeouts.

Roughly speaking, after GST, all correct processes simultaneously enter the same epoch within $O(f)$ time. After entering the same epoch, processes are guaranteed to synchronize in that epoch, which takes (at most) an additional $O(f)$ time. Thus, the latency complexity of RareSync is $O(f)$. The communication complexity of RareSync is $O(n^2)$ as every correct process executes at most a constant number of epochs, each with $O(n)$ per-process communication, after GST.

Protocol description. We now explain how RareSync works. The pseudocode of RareSync is given in Algorithm 2, whereas all variables, constants, and functions are presented in Algorithm 1.

We explain RareSync's pseudocode (Algorithm 2) from the perspective of a correct process P_i. Process P_i utilizes two timers: $view_timer_i$ and $dissemination_timer_i$. A timer has two methods:

1. measure(Time x): After exactly x time as measured by the local clock, an expiration event is received by the host. Note that, as local clocks can drift before GST, x time as measured by the local clock may not amount to x real time (before GST).
2. cancel(): This method cancels all previously invoked measure($\cdot$) methods on that timer, i.e., all pending expiration events (pertaining to that timer) are removed from the event queue.

In RareSync, leader($\cdot$) is a round-robin function (line 10 of Algorithm 1).

Once P_i starts executing RareSync (line 1), it instructs $view_timer_i$ to measure the duration of the first view (line 2) and it enters the first view (line 3).

Once $view_timer_i$ expires (line 4), P_i checks whether the current view is the last view of the current epoch, $epoch_i$ (line 5). If that is not the case, the process advances to the next view of $epoch_i$ (line 9). Otherwise, the process broadcasts an EPOCH-COMPLETED message (line 12) signaling that it has completed $epoch_i$. At this point in time, the process does not enter any view.

If, at any point in time, P_i receives either (1) $2f + 1$ EPOCH-COMPLETED messages for some epoch $e \geq epoch_i$ (line 13), or (2) an ENTER-EPOCH message for some epoch $e' > epoch_i$ (line 19), the process obtains a proof that a new epoch $E > epoch_i$ can be entered. However, before entering E and propagating the information that E can be entered, P_i waits δ time (either line 18 or line 24). This δ-waiting step is introduced to limit the number of epochs P_i can enter within any δ time period after GST and is crucial for keeping the communication complexity of RareSync quadratic. For example, suppose that processes are allowed to enter epochs and propagate ENTER-EPOCH messages without waiting. Due to an accumulation (from before GST) of ENTER-EPOCH messages for different epochs, a process might end up disseminating an arbitrary number of these messages by receiving them all at (roughly) the same time. To curb this behavior, given that message delays are bounded by δ after GST, we force a process to wait δ time, during which it receives all accumulated messages, before entering the largest known epoch.

Finally, after δ time has elapsed (line 25), P_i disseminates the information that the epoch E can be entered (line 26) and it enters the first view of E (line 30).

Algorithm 1 RareSync: Variables (for process P_i), constants, and functions.

```
1:  Variables:
2:      Epoch epoch_i ← 1                                   ▷ current epoch
3:      View view_i ← 1          ▷ current view within the current epoch; view_i ∈ [1, f + 1]
4:      Timer view_timer_i                   ▷ measures the duration of the current view
5:      Timer dissemination_timer_i ▷ measures the duration between two communication
    steps
6:      T_Signature epoch_sig_i ← ⊥                 ▷ proof that epoch_i can be entered
7:  Constants:
8:      Time view_duration = Δ + 2δ                          ▷ duration of each view
9:  Functions:
10:     leader(View v) ≡ P_{(v mod n)+1}                     ▷ a round-robin function
```

4.3 Correctness and Complexity: Proof Sketch

This subsection presents a proof sketch of the correctness, latency complexity, and communication complexity of RareSync.

In order to prove the correctness of RareSync, we must show that the eventual synchronization property is ensured, i.e., there is a synchronization time $t_s \geq GST$. For the latency complexity, it suffices to bound $t_s + \Delta - GST$ by $O(f)$. This is done by proving that synchronization happens within (at most) 2 epochs after GST. As for the communication complexity, we prove that any correct process enters a constant number of epochs during the time period $[GST, t_s + \Delta]$. Since every correct process sends $O(n)$ words per epoch, the communication complexity of RareSync is $O(n^2) = O(1) \cdot O(n) \cdot n$. We work towards these conclusions by introducing some key concepts and presenting a series of intermediate results.

A correct process *enters* an epoch e at time t if and only if the process enters the first view of e at time t (either line 3 or line 30). We denote by t_e the first time a correct process enters epoch e.

Result 1: *If a correct process enters an epoch $e > 1$, then (at least) $f + 1$ correct processes have previously entered epoch $e - 1$.*

The goal of the communication step at the end of each epoch is to prevent correct processes from arbitrarily entering future epochs. In order for a new epoch $e > 1$ to be entered, at least $f+1$ correct processes must have entered and "gone through" each view of the previous epoch, $e - 1$. This is indeed the case: in order for a correct process to enter e, the process must either (1) collect $2f + 1$ epoch-completed messages for $e - 1$ (line 13), or (2) receive an enter-epoch message for e, which contains a threshold signature of $e - 1$ (line 19). In either case, at least $f + 1$ correct processes must have broadcast epoch-completed messages for epoch $e - 1$ (line 12), which requires them to go through epoch $e - 1$. Furthermore, $t_{e-1} \leq t_e$; recall that local clocks can drift before GST.

Result 2: *Every epoch is eventually entered by a correct process.*

By contradiction, consider the greatest epoch ever entered by a correct process, e^*. In brief, every correct process will eventually (1) receive the enter-epoch message for e^* (line 19), (2) enter e^* after its *dissemination_timer* expires (lines 25 and 30), (3) send an epoch-completed message for e^* (line 12), (4) collect $2f + 1$ epoch-completed messages for e^* (line 13), and, finally, (5) enter $e^* + 1$ (lines 15, 18, 25 and 30), resulting in a contradiction. Note that, if $e^* = 1$, no enter-epoch message is sent: all correct processes enter $e^* = 1$ once they start executing RareSync (line 3).

We now define two epochs: e_{max} and $e_{final} = e_{max} + 1$. These two epochs are the main protagonists in the proof of correctness and complexity of RareSync.

Definition of e_{max}: *Epoch e_{max} is the greatest epoch entered by a correct process before GST; if no such epoch exists, $e_{max} = 0$.*[4]

Definition of e_{final}: *Epoch e_{final} is the smallest epoch first entered by a correct process at or after GST. Note that $GST \leq t_{e_{final}}$. Moreover, $e_{final} = e_{max} + 1$ (by Result 1).*

Result 3: *For any epoch $e \geq e_{final}$, no correct process broadcasts an epoch-completed message for e (line 12) before time $t_e + epoch_duration$, where $epoch_duration = (f + 1) \cdot view_duration$.*

[4] Epoch 0 is considered as a special epoch. Note that $0 \notin \mathsf{Epoch}$, where Epoch denotes the set of epochs (see §4.2).

This statement is a direct consequence of the fact that, after GST, it takes exactly $epoch_duration$ time for a process to go through $f+1$ views of an epoch; local clocks do not drift after GST. Specifically, the earliest a correct process can broadcast an EPOCH-COMPLETED message for e (line 12) is at time $t_e + epoch_duration$, where t_e denotes the first time a correct process enters epoch e.

Result 4: *Every correct process enters epoch e_{final} by time $t_{e_{final}} + 2\delta$.*

Recall that the first correct process enters e_{final} at time $t_{e_{final}}$. If $e_{final} = 1$, all correct processes enter e_{final} at $t_{e_{final}}$. Otherwise, by time $t_{e_{final}} + \delta$, all correct processes will have received an ENTER-EPOCH message for e_{final} and started the $dissemination_timer_i$ with $epoch_i = e_{final}$ (either lines 15, 18 or 21, 24). By results 1 and 3, no correct process sends an EPOCH-COMPLETED message for an epoch $\geq e_{final}$ (line 12) before time $t_{e_{final}} + epoch_duration$, which implies that the $dissemination_timer$ will not be cancelled. Hence, the $dissemination_timer$ will expire by time $t_{e_{final}} + 2\delta$, causing all correct processes to enter e_{final} by time $t_{e_{final}} + 2\delta$.

Result 5: *In every view of e_{final}, processes overlap for (at least) Δ time. In other words, there exists a synchronization time $t_s \leq t_{e_{final}} + epoch_duration - \Delta$.*

By Result 3, no future epoch can be entered before time $t_{e_{final}} + epoch_duration$. This is precisely enough time for the first correct process (the one to enter e_{final} at $t_{e_{final}}$) to go through all $f+1$ views of e_{final}, spending $view_duration$ time in each view. Since clocks do not drift after GST and processes spend the same amount of time in each view, the maximum delay of 2δ between processes (Result 4) applies to every view in e_{final}. Thus, all correct processes overlap with each other for (at least) $view_duration - 2\delta = \Delta$ time in every view of e_{final}. As the $\mathsf{leader}(\cdot)$ function is round-robin, at least one of the $f+1$ views must have a correct leader. Therefore, synchronization must happen within epoch e_{final}, i.e., there is a synchronization time t_s such that $t_{e_{final}} + \Delta \leq t_s + \Delta \leq t_{e_{final}} + epoch_duration$.

Result 6: $t_{e_{final}} \leq GST + epoch_duration + 4\delta$.

If $e_{final} = 1$, all correct processes started executing RARESYNC at time GST. Hence, $t_{e_{final}} = GST$. Therefore, the result trivially holds in this case.

Let $e_{final} > 1$; recall that $e_{final} = e_{max} + 1$. (1) By time $GST + \delta$, every correct process receives an ENTER-EPOCH message for e_{max} (line 19) as the first correct process to enter e_{max} has broadcast this message before GST (line 26). Hence, (2) by time $GST + 2\delta$, every correct process enters e_{max}.[5] Then, (3) every correct process broadcasts an EPOCH-COMPLETED message for e_{max} at time $GST + epoch_duration + 2\delta$ (line 12), at latest. (4) By time $GST + epoch_duration + 3\delta$, every correct process receives $2f+1$ EPOCH-COMPLETED messages for e_{max} (line 13), and triggers the $\mathsf{measure}(\delta)$ method of $dissemination_timer$ (line 18). Therefore, (5) by time $GST + epoch_duration + 4\delta$, every correct process enters $e_{max} + 1 = e_{final}$. Figure 2 depicts this scenario.

Note that for the previous sequence of events *not* to unfold would imply an even lower bound on $t_{e_{final}}$: a correct process would have to receive $2f+1$ EPOCH-COMPLETED messages for e_{max} or an ENTER-EPOCH message for $e_{max} + 1 = e_{final}$ before step (4) (i.e., before time $GST + epoch_duration + 3\delta$), thus showing that $t_{e_{final}} < GST + epoch_duration + 4\delta$.

Latency: *Latency complexity of RARESYNC is $O(f)$.*

By Result 5, $t_s \leq t_{e_{final}} + epoch_duration - \Delta$. By Result 6, $t_{e_{final}} \leq GST + epoch_duration + 4\delta$. Therefore, $t_s \leq GST + epoch_duration + 4\delta + epoch_duration - \Delta = GST + 2epoch_duration + 4\delta - \Delta$. Hence, $t_s + \Delta - GST \leq 2epoch_duration + 4\delta = O(f)$.

[5] If $e_{max} = 1$, every correct process enters e_{max} by time GST.

Communication: *Communication complexity of* RARESYNC *is* $O(n^2)$.

Roughly speaking, every correct process will have entered e_{max} (or potentially $e_{final} = e_{max}+1$) by time $GST + 2\delta$ (as seen in the proof of Result 6). From then on, it will enter at most one other epoch (e_{final}) before synchronizing (which is completed by time $t_s + \Delta$). As for the time interval $[GST, GST + 2\delta)$, due to *dissemination_timer*'s interval of δ, a correct process can enter (at most) two other epochs during this period. Therefore, a correct process can enter (and send messages for) at most $O(1)$ epochs between GST and $t_s + \Delta$. The individual communication cost of a correct process is bounded by $O(n)$ words per epoch: $O(n)$ EPOCH-COMPLETED messages (each with a single word), and $O(n)$ ENTER-EPOCH messages (each with a single word, as a threshold signature counts as a single word). Thus, the communication complexity of RARESYNC is $O(n^2) = O(1) \cdot O(n) \cdot n$.

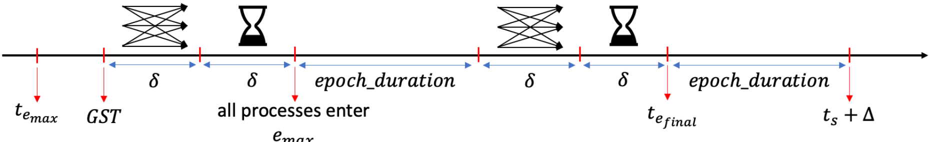


Figure 2 Worst-case latency of RARESYNC: $t_s + \Delta - GST \leq 2epoch_duration + 4\delta$.

▶ **Theorem 2.** *RARESYNC is a partially synchronous view synchronizer with (1) $O(n^2)$ communication complexity, and (2) $O(f)$ latency complexity.*

5 SQuad

This section introduces SQUAD, a partially synchronous Byzantine consensus protocol with optimal resilience [26]. SQUAD simultaneously achieves (1) $O(n^2)$ communication complexity, matching the Dolev-Reischuk bound [25], and (2) $O(f)$ latency complexity, matching the Dolev-Strong bound [23].

First, we present QUAD, a partially synchronous Byzantine consensus protocol ensuring weak validity (§5.1). QUAD achieves quadratic communication complexity and linear latency complexity. Then, we construct SQUAD by adding a simple preprocessing phase to QUAD (§5.2).

5.1 Quad

QUAD is a partially synchronous Byzantine consensus protocol satisfying the weak validity property:

- *Weak validity:* If all processes are correct, then a value decided by a process was proposed.

QUAD achieves (1) quadratic communication complexity, and (2) linear latency complexity. Interestingly, the Dolev-Reischuk lower bound [25] does not apply to Byzantine protocols satisfying weak validity; hence, we do not know whether QUAD has optimal communication complexity. As explained in §5.2, we accompany QUAD by a preprocessing phase to obtain SQUAD.

QUAD (Algorithm 3) uses the same view core module as HotStuff [56], i.e., the view logic of QUAD is identical to that of HotStuff. Moreover, QUAD uses RARESYNC as its view synchronizer, achieving synchronization with $O(n^2)$ communication. The combination of HotStuff's view core and RARESYNC ensures that each correct process sends $O(n)$ words after GST (and before the decision), i.e., $C = O(n)$ in QUAD. Following the formula introduced in §1, QUAD indeed achieves $n \cdot C + S = n \cdot O(n) + O(n^2) = O(n^2)$ communication complexity. Due to the linear latency of RARESYNC, QUAD also achieves $O(f)$ latency complexity.

View core. We now give a brief description of the view core module of Quad. The complete pseudocode of this module can be found in [56].

Each correct process keeps track of two critical variables: (1) the *prepare* quorum certificate (QC), and (2) the *locked* QC. Each of these represents a process' estimation of the value that will be decided, although with a different degree of certainty. For example, if a correct process decides a value v, it is guaranteed that (at least) $f + 1$ correct processes have v in their locked QC. Moreover, it is ensured that no correct process updates (from this point onward) its prepare or locked QC to any other value, thus ensuring agreement. Lastly, a QC is a (constant-sized) threshold signature.

The structure of a view follows the "all-to-leader, leader-to-all" communication pattern. Specifically, each view is comprised of the following four phases:

1. **Prepare:** A process sends to the leader a view-change message containing its prepare QC. Once the leader receives $2f + 1$ view-change messages, it selects the prepare QC from the "latest" view. The leader sends this QC to all processes via a prepare message. Once a process receives the prepare message from the leader, it supports the received prepare QC if (1) the received QC is consistent with its locked QC, or (2) the received QC is "more recent" than its locked QC. If the process supports the received QC, it acknowledges this by sending a prepare-vote message to the leader.
2. **Precommit:** Once the leader receives $2f + 1$ prepare-vote messages, it combines them into a cryptographic proof σ that "enough" processes have supported its "prepare-phase" value; σ is a threshold signature. Then, it disseminates σ to all processes via a precommit message. Once a process receives the precommit message carrying σ, it updates its prepare QC to σ and sends back to the leader a precommit-vote message.
3. **Commit:** Once the leader receives $2f + 1$ precommit-vote messages, it combines them into a cryptographic proof σ' that "enough" processes have adopted its "precommit-phase" value (by updating their prepare QC); σ' is a threshold signature. Then, it disseminates σ' to all processes via a commit message. Once a process receives the commit message carrying σ', it updates its locked QC to σ' and sends back to the leader a commit-vote message.
4. **Decide:** Once the leader receives $2f + 1$ commit-vote messages, it combines them into a threshold signature σ'', and relays σ'' to all processes via a decide message. When a process receives the decide message carrying σ'', it decides the value associated with σ''.

As a consequence of the "all-to-leader, leader-to-all" communication pattern and the constant size of messages, the leader of a view sends $O(n)$ words, while a non-leader process sends $O(1)$ words.

The view core module provides the following interface:

- **Request** start_executing(View v): The view core starts executing the logic of view v and abandons the previous view. Concretely, it stops accepting and sending messages for the previous view, and it starts accepting, sending, and replying to messages for view v. The state of the view core is kept across views (e.g., the prepare and locked QCs).
- **Indication** decide(Value *decision*): The view core decides value *decision* (this indication is triggered at most once).

Protocol description. The protocol (Algorithm 3) amounts to a composition of RareSync and the aforementioned view core. Since the view core requires 8 communication steps in order for correct processes to decide, a synchronous overlap of 8δ is sufficient. Thus, we parameterize RareSync with $\Delta = 8\delta$ (line 3). In short, the view core is subservient to

RareSync, i.e., when RareSync triggers the advance(v) event (line 7), the view core starts executing the logic of view v (line 8). Once the view core decides (line 9), Quad decides (line 10).

Proof sketch. The agreement and weak validity properties of Quad are ensured by the view core's implementation. As for the termination property, the view core, and therefore Quad, is guaranteed to decide as soon as processes have synchronized in the same view with a correct leader for $\Delta = 8\delta$ time at or after GST. Since RareSync ensures the eventual synchronization property, this eventually happens, which implies that Quad satisfies termination. As processes synchronize within $O(f)$ time after GST, the latency complexity of Quad is $O(f)$.

As for the total communication complexity, it is the sum of the communication complexity of (1) RareSync, which is $O(n^2)$, and (2) the view core, which is also $O(n^2)$. The view core's complexity is a consequence of the fact that:

- each process executes $O(1)$ epochs between GST and the time by which every process decides,
- each epoch has $f + 1$ views,
- a process can be the leader in only one view of any epoch, and
- a process sends $O(n)$ words in a view if it is the leader, and $O(1)$ words otherwise, for an average of $O(1)$ words per view in any epoch.

Thus, the view core's communication complexity is $O(n^2) = O(1) \cdot (f + 1) \cdot O(1) \cdot n$. Therefore, Quad indeed achieves $O(n^2)$ communication complexity. ◀

▶ **Theorem 3.** *Quad is a Byzantine consensus protocol ensuring weak validity with (1) $O(n^2)$ communication complexity, and (2) $O(f)$ latency complexity.*

5.2 SQuad: Protocol Description

At last, we present SQuad, which we derive from Quad.

Deriving SQuad from Quad. Imagine a locally-verifiable, *constant-sized* cryptographic proof σ_v vouching that value v is *valid*. Moreover, imagine that it is impossible, in the case in which all correct processes propose v to Quad, for any process to obtain a proof for a value different from v:

- Computability: If all correct processes propose v to Quad, then no process (even if faulty) obtains a cryptographic proof $\sigma_{v'}$ for a value $v' \neq v$.

If such a cryptographic primitive were to exist, then the Quad protocol could be modified in the following manner in order to satisfy the validity property introduced in §1:

- A correct process accompanies each value by a cryptographic proof that the value is valid.
- A correct process ignores any message with a value not accompanied by the value's proof.

Suppose that all correct processes propose the same value v and that a correct process P_i decides v' from the modified version of Quad. Given that P_i ignores messages with non-valid values, P_i has obtained a proof for v' before deciding. The computability property of the cryptographic primitive guarantees that $v' = v$, implying that validity is satisfied. Given that the proof is of constant size, the communication complexity of the modified version of Quad remains $O(n^2)$.

Therefore, the main challenge in obtaining SQuad from Quad, while preserving Quad's complexity, lies in implementing the introduced cryptographic primitive.

Certification phase. SQUAD utilizes its *certification phase* (Algorithm 4) to obtain the introduced constant-sized cryptographic proofs; we call these proofs *certificates*.[6] Formally, Certificate denotes the set of all certificates. Moreover, we define a locally computable function verify: Value × Certificate → $\{true, false\}$. We require the following properties to hold:

- *Computability:* If all correct processes propose the same value v to SQUAD, then no process (even if faulty) obtains a certificate $\sigma_{v'}$ with $\mathsf{verify}(v', \sigma_{v'}) = true$ and $v' \neq v$.
- *Liveness:* Every correct process eventually obtains a certificate σ_v such that $\mathsf{verify}(v, \sigma_v) = true$, for some value v.

The computability property states that, if all correct processes propose the same value v to SQUAD, then no process (even if Byzantine) can obtain a certificate for a value different from v. The liveness property ensures that all correct processes eventually obtain a certificate. Hence, if all correct processes propose the same value v, all correct processes eventually obtain a certificate for v and no process obtains a certificate for a different value.

In order to implement the certification phase, we assume an $(f+1, n)$-threshold signature scheme (see §3) used throughout the entirety of the certification phase. The $(f+1, n)$-threshold signature scheme allows certificates to count as a single word, as each certificate is a threshold signature. Finally, in order to not disrupt QUAD's communication and latency, the certification phase itself incurs $O(n^2)$ communication and $O(1)$ latency.

A certificate σ vouches for a value v (the $\mathsf{verify}(\cdot)$ function at line 21) if (1) σ is a threshold signature of the predefined string "any value" (line 22), or (2) σ is a threshold signature of v (line 23). Otherwise, $\mathsf{verify}(v, \sigma)$ returns *false*.

Once P_i enters the certification phase (line 1), P_i informs all processes about the value it has proposed by broadcasting a DISCLOSE message (line 3). Process P_i includes a partial signature of its proposed value in the message. If P_i receives DISCLOSE messages for the same value v from $f+1$ processes (line 4), P_i combines the received partial signatures into a threshold signature of v (line 6), which represents a certificate for v. To ensure liveness, P_i disseminates the certificate (line 7).

If P_i receives $2f+1$ DISCLOSE messages and there does not exist a "common" value received in $f+1$ (or more) DISCLOSE messages (line 9), the process concludes that it is fine for a certificate for *any* value to be obtained. Therefore, P_i broadcasts an ALLOW-ANY message containing a partial signature of the predefined string "any value" (line 11).

If P_i receives $f+1$ ALLOW-ANY messages (line 12), it combines the received partial signatures into a certificate that vouches for *any* value (line 14), and it disseminates the certificate (line 15). Since ALLOW-ANY messages are received from $f+1$ processes, there exists a correct process that has verified that it is indeed fine for such a certificate to exist.

If, at any point, P_i receives a certificate (line 18), it adopts the certificate, and disseminates it (line 19) to ensure liveness.

Given that each message of the certification phase contains a single word, the certification phase incurs $O(n^2)$ communication. Moreover, each correct process obtains a certificate after (at most) $2 = O(1)$ rounds of communication. Therefore, the certification phase incurs $O(1)$ latency.

We explain below why the certification phase (Algorithm 4) ensures computability and liveness:

- Computability: If all correct processes propose the same value v to SQUAD, all correct processes broadcast a DISCLOSE message for v (line 3). Since $2f+1$ processes are correct, no process obtains a certificate $\sigma_{v'}$ for a value $v' \neq v$ such that $\mathit{CombinedVerify}(v', \sigma_{v'}) = true$ (line 23).

[6] Note the distinction between certificates and prepare and locked QCs of the view core.

Moreover, as every correct process receives $f+1$ DISCLOSE messages for v within any set of $2f+1$ received DISCLOSE messages, no correct process sends an ALLOW-ANY message (line 11). Hence, no process obtains a certificate $\sigma_\perp$ such that *CombinedVerify*(“any value”, $\sigma_\perp$) = *true* (line 22). Thus, computability is ensured.

- Liveness: If a correct process receives $f+1$ DISCLOSE messages for a value v (line 4), the process obtains a certificate for v (line 6). Since the process disseminates the certificate (line 7), every correct process eventually obtains a certificate (line 18), ensuring liveness in this scenario.
  Otherwise, all correct processes broadcast an ALLOW-ANY message (line 11). Since there are at least $2f+1$ correct processes, every correct process eventually receives $f+1$ ALLOW-ANY messages (line 12), thus obtaining a certificate. Hence, liveness is satisfied in this case as well.

SQuad = Certification phase + Quad. We obtain SQUAD by combining the certification phase with QUAD. The pseudocode of SQUAD is given in Algorithm 5.

A correct process P_i executes the following steps in SQUAD:

1. P_i starts executing the certification phase with its proposal (line 2).
2. Once the process exits the certification phase with a certificate σ_v for a value v, it proposes (v, σ_v) to QUAD$_{cer}$, a version of QUAD “enriched” with certificates (line 5). While executing QUAD$_{cer}$, correct processes *ignore* messages containing values not accompanied by their certificates.
3. Once P_i decides from QUAD$_{cer}$ (line 6), P_i decides the same value from SQUAD (line 7).

▶ **Theorem 4.** *SQUAD is a Byzantine consensus protocol with (1) $O(n^2)$ communication complexity, and (2) $O(f)$ latency complexity.*

6 Concluding Remarks

This paper shows that the Dolev-Reischuk lower bound can be met by a partially synchronous Byzantine consensus protocol. Namely, we introduce SQUAD, an optimally-resilient partially synchronous Byzantine consensus protocol with optimal $O(n^2)$ communication complexity, and optimal $O(f)$ latency complexity. SQUAD owes its complexity to RARESYNC, an “epoch-based” view synchronizer ensuring synchronization with quadratic communication and linear latency in partial synchrony.

References

1. Ittai Abraham, T-H. Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Communication Complexity of Byzantine Agreement, Revisited. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC ’19, pages 317–326, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293611.3331629.
2. Ittai Abraham, Srinivas Devadas, Kartik Nayak, and Ling Ren. Brief Announcement: Practical Synchronous Byzantine Consensus. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 41:1–41:4. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.DISC.2017.41.
3. Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching Consensus for Asynchronous Distributed Key Generation. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC ’21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 363–373. ACM, 2021. doi:10.1145/3465084.3467914.

4 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solida: A Blockchain Protocol Based on Reconfigurable Byzantine Consensus. In James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão, editors, *21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal, December 18-20, 2017*, volume 95 of *LIPIcs*, pages 25:1–25:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.OPODIS.2017.25.

5 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically Optimal Validated Asynchronous Byzantine Agreement. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 337–346, 2019.

6 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically Optimal Validated Asynchronous Byzantine Agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 337–346, 2019.

7 Marcin Andrychowicz and Stefan Dziembowski. PoW-Based Distributed Cryptography with No Trusted Setup. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology – CRYPTO 2015 – 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 379–399. Springer, 2015. doi:10.1007/978-3-662-48000-7_19.

8 Karolos Antoniadis, Antoine Desjardins, Vincent Gramoli, Rachid Guerraoui, and Igor Zablotchi. Leaderless Consensus. In *Proceedings – International Conference on Distributed Computing Systems*, volume 2021-July, pages 392–402, 2021.

9 Michael Ben-Or. Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. *Proceedings of the Second Annual Symposium on Principles of Distributed Computing*, pages 27–30, 1983.

10 Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Bit Optimal Distributed Consensus. *Computer Science: Research and Applications*, pages 313–321, 1992.

11 Gabriel Bracha. Asynchronous Byzantine Agreement Protocols. *Inf. Comput.*, 75(2):130–143, 1987. doi:10.1016/0890-5401(87)90054-X.

12 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making Byzantine Consensus Live. In *34th International Symposium on Distributed Computing (DISC)*, volume 179(23), pages 1–17, 2020.

13 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, pages 1–14, 2018. arXiv:1807.04938.

14 Christian Cachin, Klaus Kursawe, and Victor Shoup. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. *J. Cryptol.*, 18(3):219–246, 2005. doi:10.1007/s00145-005-0318-0.

15 Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. *ACM Trans. Comput. Syst.*, pages 359–368, 2002.

16 Tushar Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pages 225–267, 1996.

17 Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The Weakest Failure Detector for Solving Consensus. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, 43(4):147–158, 1992.

18 Jing Chen, Sergey Gorbunov, Silvio Micali, and Georgios Vlachos. Algorand Agreement: Super Fast and Partition Resilient Byzantine Agreement. *Cryptology ePrint Archive*, 377:1–10, 2018. URL: https://eprint.iacr.org/2018/377.pdf.

19 Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. Byzantine Consensus is $\Theta(n^2)$: The Dolev-Reischuk Bound is Tight even in Partial Synchrony! [Extended Version], 2022. doi:10.48550/ARXIV.2208.09262.

20 Shir Cohen, Idit Keidar, and Oded Naor. Byzantine Agreement with Less Communication: Recent Advances. *SIGACT News*, 52(1):71–80, 2021. doi:10.1145/3457588.3457600.

21 Shir Cohen, Idit Keidar, and Alexander Spiegelman. Brief Announcement: Not a COINcidence: Sub-Quadratic Asynchronous Byzantine Agreement WHP. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 175–177, 2020.

22 Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: Efficient Byzantine Consensus with a Weak Coordinator and its Application to Consortium Blockchains. In *17th IEEE International Symposium on Network Computing and Applications, NCA*, pages 1–41, 2017. arXiv:1702.03068.

23 D. Dolev and H. R. Strong. Authenticated Algorithms for Byzantine Agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

24 Danny Dolev, Joseph Y. Halpern, Barbara Simons, and Ray Strong. Dynamic Fault-Tolerant Clock Synchronization. *Journal of the ACM (JACM)*, 42(1):143–185, 1995.

25 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for Byzantine agreement. *Journal of the ACM (JACM)*, 1985.

26 Cynthia Dwork, Lynch Nancy, and Larry Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

27 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the Association for Computing Machinery,*, 32(2):374–382, 1985.

28 Eli Gafni. Round-by-Round Fault Detectors: Unifying Synchrony and Asynchrony. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 143–152, 1998.

29 Juan A. Garay, Aggelos Kiayias, Nikos Leonardos, and Giorgos Panagiotakos. Bootstrapping the Blockchain, with Applications to Consensus and Fast PKI Setup. In Michel Abdalla and Ricardo Dahab, editors, *Public-Key Cryptography – PKC 2018 – 21st IACR International Conference on Practice and Theory of Public-Key Cryptography, Rio de Janeiro, Brazil, March 25-29, 2018, Proceedings, Part II*, volume 10770 of *Lecture Notes in Computer Science*, pages 465–495. Springer, 2018. doi:10.1007/978-3-319-76581-5_16.

30 Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A Scalable and Decentralized Trust Infrastructure. *Proceedings – 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019*, pages 568–580, 2019.

31 Vincent Gramoli. From blockchain consensus back to Byzantine consensus. *Future Gener. Comput. Syst.*, 107:760–769, 2020. doi:10.1016/j.future.2017.09.023.

32 Rachid Guerraoui and Michel Raynal. The Information Structure of Indulgent Consensus. *IEEE Trans. Computers*, 53(4):453–466, 2004.

33 Idit Keidar and Alexander Shraer. Timeliness, Failure-Detectors, and Consensus Performance. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, 2006:169–178, 2006.

34 Kim Potter Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith. Byzantine Fault Detectors for Solving Consensus. *The Computer Journal*, 46(1):16–35, 2003.

35 Valerie King and Jared Saia. Breaking the $O(n^2)$ Bit Barrier: Scalable Byzantine agreement with an Adaptive Adversary. *Journal of the ACM*, 58(4):1–24, 2011.

36 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems*, 27(4), 2009.

37 Petr Kuznetsov, Andrei Tonkikh, and Yan X. Zhang. Revisiting Optimal Resilience of Fast Byzantine Consensus. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1(1):343–353, 2021.

38 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

39 Andrew Lewis-Pye. Quadratic worst-case message complexity for State Machine Replication in the partial synchrony model, 2022. doi:10.48550/ARXIV.2201.01107.

40 Benoît Libert, Marc Joye, and Moti Yung. Born and Raised Distributively: Fully Distributed Non-Interactive Adaptively-Secure Threshold Signatures with Short Shares. *Theoretical Computer Science*, 645:1–24, 2016.

41 JongBeom Lim, Taeweon Suh, Joon-Min Gil, and Heon-Chang Yu. Scalable and leaderless Byzantine consensus in cloud computing environments. *Inf. Syst. Frontiers*, 16(1):19–34, 2014. doi:10.1007/s10796-013-9460-7.

42 Thomas Locher. Fast Byzantine Agreement for Permissioned Distributed Ledgers. *Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 371–382, 2020.

43 Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-MVBA: Optimal Multi-Valued Validated Asynchronous Byzantine Agreement, Revisited. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 129–138, 2020.

44 Jean Philippe Martin and Lorenzo Alvisi. Fast Byzantine Consensus. *Proceedings of the International Conference on Dependable Systems and Networks*, pages 402–411, 2005.

45 Silvio Micali. Byzantine Agreement , Made Trivial, 2017.

46 Atsuki Momose and Ling Ren. Optimal Communication Complexity of Authenticated Byzantine Agreement. In *35th International Symposium on Distributed Computing (DISC)*, volume 209(32), pages 32:1–32:0. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany, 2021.

47 Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-Free Asynchronous Binary Byzantine Consensus with t < n/3, O(n2) Messages, and O(1) Expected Time. *J. ACM*, 62(4):31:1–31:21, 2015. doi:10.1145/2785953.

48 Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine View Synchronization. *Cryptoeconomic Systems*, 2021.

49 Oded Naor and Idit Keidar. Expected Linear Round Synchronization: The Missing Link for Linear Byzantine SMR. *34th International Symposium on Distributed Computing (DISC)*, 179, 2020.

50 Rafael Pass and Elaine Shi. Thunderella: Blockchains with Optimistic Instant Confirmation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10821 LNCS:3–33, 2018.

51 Michael O. Rabin. Randomized Byzantine Generals. In *24th Annual Symposium on Foundations of Computer Science, Tucson, Arizona, USA, 7-9 November 1983*, pages 403–409. IEEE Computer Society, 1983. doi:10.1109/SFCS.1983.48.

52 Hari Govind V. Ramasamy and Christian Cachin. Parsimonious Asynchronous Byzantine-Fault-Tolerant Atomic Broadcast. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3974 LNCS:88–102, 2006.

53 Alexander Spiegelman. In Search for an Optimal Authenticated Byzantine Agreement. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DISC.2021.38.

54 T. K. Srikanth and Sam Toueg. Optimal Clock Synchronization. *Journal of the Association for Computing Machinery*, 34(3):71–86, 1987.

55 The Diem Team. DiemBFT v4: State Machine Replication in the Diem Blockchain, 2021. URL: https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf.

56 Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus with Linearity and Responsiveness. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

Algorithm 2 RARESYNC: Pseudocode (for process P_i).

1: **upon** init: ▷ start of the protocol
2: $view_timer_i$.measure($view_duration$) ▷ measure the duration of the first view
3: **trigger** advance(1) ▷ enter the first view
4: **upon** $view_timer_i$ **expires**:
5: **if** $view_i < f + 1$: ▷ check if the current view is not the last view of the current epoch
6: $view_i \leftarrow view_i + 1$
7: View $view_to_advance \leftarrow (epoch_i - 1) \cdot (f + 1) + view_i$
8: $view_timer_i$.measure($view_duration$) ▷ measure the duration of the view
9: **trigger** advance($view_to_advance$) ▷ enter the next view
10: **else:**
11: ▷ inform other processes that the epoch is completed
12: **broadcast** ⟨EPOCH-COMPLETED, $epoch_i$, $ShareSign_i(epoch_i)$⟩
13: **upon** exists Epoch e such that $e \geq epoch_i$ and ⟨EPOCH-COMPLETED, e, P_Signature sig⟩ is received from $2f + 1$ processes:
14: $epoch_sig_i \leftarrow Combine(\{sig \mid sig$ is received in an EPOCH-COMPLETED message$\})$
15: $epoch_i \leftarrow e + 1$
16: $view_timer_i$.cancel()
17: $dissemination_timer_i$.cancel()
18: $dissemination_timer_i$.measure(δ) ▷ wait δ time before broadcasting ENTER-EPOCH
19: **upon** reception of ⟨ENTER-EPOCH, Epoch e, T_Signature sig⟩ such that $e > epoch_i$:
20: $epoch_sig_i \leftarrow sig$ ▷ sig is a threshold signature of epoch $e - 1$
21: $epoch_i \leftarrow e$
22: $view_timer_i$.cancel()
23: $dissemination_timer_i$.cancel()
24: $dissemination_timer_i$.measure(δ) ▷ wait δ time before broadcasting ENTER-EPOCH
25: **upon** $dissemination_timer_i$ **expires**:
26: **broadcast** ⟨ENTER-EPOCH, $epoch_i$, $epoch_sig_i$⟩
27: $view_i \leftarrow 1$ ▷ reset the current view to 1
28: View $view_to_advance \leftarrow (epoch_i - 1) \cdot (f + 1) + view_i$
29: $view_timer_i$.measure($view_duration$) ▷ measure the duration of the view
30: **trigger** advance($view_to_advance$) ▷ enter the first view of the new epoch

Algorithm 3 QUAD: Pseudocode (for process P_i).

1: **Modules:**
2: View_Core $core$
3: View_Synchronizer $synchronizer \leftarrow$ RARESYNC($\Delta = 8\delta$)
4: **upon** init(Value $proposal$): ▷ propose value $proposal$
5: $core$.init($proposal$) ▷ initialize the view core with the proposal
6: $synchronizer$.init ▷ start RARESYNC
7: **upon** $synchronizer$.advance(View v):
8: $core$.start_executing(v)
9: **upon** $core$.decide(Value $decision$):
10: **trigger** decide($decision$) ▷ decide value $decision$

Algorithm 4 Certification Phase: Pseudocode (for process P_i).

```
1: upon init(Value proposal):                                        ▷ propose value proposal
2:     ▷ inform other processes that proposal was proposed
3:     broadcast ⟨DISCLOSE, proposal, ShareSign_i(proposal)⟩
4: upon exists Value v such that ⟨DISCLOSE, v, P_Signature sig⟩ is received from f + 1
   processes:
5:     ▷ a certificate for v is obtained
6:     Certificate σ_v ← Combine({sig | sig is received in a DISCLOSE message})
7:     broadcast ⟨CERTIFICATE, v, σ_v⟩                                 ▷ disseminate the certificate
8:     exit the certification phase
9: upon for the first time (1) DISCLOSE message is received from 2f + 1 processes, and
   (2) not exist Value v such that ⟨DISCLOSE, v, P_Signature sig⟩ is received from f + 1
   processes:
10:    ▷ inform other processes that any value can be "accepted"
11:    broadcast ⟨ALLOW-ANY, ShareSign_i("any value")⟩
12: upon ⟨ALLOW-ANY, P_Signature sig⟩ is received from f + 1 processes :
13:    ▷ a certificate for "any value" is obtained
14:    Certificate σ_⊥ ← Combine({sig | sig is received in an ALLOW-ANY message})
15:    broadcast ⟨CERTIFICATE, ⊥, σ_⊥⟩                                 ▷ disseminate the certificate
16:    exit the certification phase
17: ▷ a certificate for v is obtained; v can be ⊥, meaning that σ_v vouches for any value
18: upon reception of ⟨CERTIFICATE, Value v, Certificate σ_v⟩:
19:    broadcast ⟨CERTIFICATE, v, σ_v⟩                                  ▷ disseminate the certificate
20:    exit the certification phase
21: function verify(Value v, Certificate σ):
22:    if CombinedVerify("any value", σ) = true: return true
23:    else if CombinedVerify(v, σ) = true: return true
24:    else return false
```

Algorithm 5 SQUAD: Pseudocode (for process P_i).

```
1: upon init(Value proposal):                                        ▷ propose value proposal
2:     start the certification phase with proposal
3: upon exiting the certification phase with a certificate σ_v for a value v:
4:     ▷ in QUAD_cer, processes ignore messages with values not accompanied by their
   certificates
5:     start executing QUAD_cer with the proposal (v, σ_v)
6: upon QUAD_cer decides Value decision:
7:     trigger decide(decision)                                      ▷ decide value decision
```

Dynamic Probabilistic Input Output Automata

Pierre Civit
Sorbonne Université, CNRS, LIP6, Paris, France

Maria Potop-Butucaru
Sorbonne Université, CNRS, LIP6, Paris, France

Abstract

We present *probabilistic dynamic I/O automata*, a framework to model dynamic probabilistic systems. Our work extends *dynamic I/O Automata* formalism of Attie & Lynch [2] to the probabilistic setting. The original dynamic I/O Automata formalism included operators for parallel composition, action hiding, action renaming, automaton creation, and behavioral sub-typing by means of trace inclusion. They can model mobility by using signature modification. They are also hierarchical: a dynamically changing system of interacting automata is itself modeled as a single automaton. Our work extends all these features to the probabilistic setting. Furthermore, we prove necessary and sufficient conditions to obtain the monotonicity of automata creation/destruction with implementation preorder. Our construction uses a novel proof technique based on homomorphism that can be of independent interest. Our work lays down the foundations for extending *composable secure-emulation* of Canetti et al. [5] to dynamic settings, an important tool towards the formal verification of protocols combining probabilistic distributed systems and cryptography in dynamic settings (e.g. blockchains, secure distributed computation, cybersecure distributed protocols, etc).

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Automata, Distributed Computing, Formal Verification, Dynamic systems

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.15

Related Version *Full Version*: https://eprint.iacr.org/2021/798

1 Introduction

Distributed computing area faces today important challenges coming from modern applications such as peer-to-peer networks, cooperative robotics, dynamic sensor networks, adhoc networks and more recently, cryptocurrencies and blockchains which have a tremendous impact in our society. These newly emerging fields of distributed systems are characterized by an extreme dynamism in terms of structure, content and load. Moreover, they have to offer strong guaranties over large scale networks which is usually impossible in deterministic settings. Therefore, most of these systems use probabilistic algorithms and randomized techniques in order to offer scalability features. However, the vulnerabilities of these systems may be exploited with the aim to provoke an unforeseen execution that diverges from the understanding or intuition of the developers. Therefore, formal validation and verification of these systems has to be realized before their industrial deployment.

It is difficult to attribute the pioneering of formalization of concurrent systems to some particular authors [16, 10, 1, 15, 11, 13, 9]. Lynch and Tuttle [12] proposed the formalism of *Input/Output Automata* to model deterministic asynchronous distributed systems. Relationship between process algebra and I/O automata are discussed in [20, 14]. Later, this formalism is extended by Segala in [19] with Markov decision processes [17]. In order to model randomized distributed systems Segala proposes *Probabilistic Input/Output Automata*. In this model each process in the system is an automaton with probabilistic transitions. The probabilistic protocol is the parallel composition of the automata modeling each participant.

© Pierre Civit and Maria Potop-Butucaru;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 15; pp. 15:1–15:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The modelisation of dynamic behavior in distributed systems has been addressed by Attie & Lynch in [2] where they propose *Dynamic Input Output Automata* formalism. This formalism extends the *Input/Output Automata* with the ability to change their signature dynamically (i.e. the set of actions in which the automaton can participate) and to create other I/O automata or destroy existing I/O automata. The formalism introduced in [2] does not cover the case of probabilistic distributed systems and therefore cannot be used in the verification of recent blockchains such as Algorand [6].

In order to respond to the need of formalisation in secure distributed systems, Canetti & al. proposed in [3] *task-structured probabilistic Input/Output automata* (TPIOA) specifically designed for the analysis of cryptographic protocols. Task-structured probabilistic Input/Output automata are Probabilistic Input/Output automata extended with tasks that are equivalence classes on the set of actions. The task-structure allows a generalisation of "off-line scheduling" where the non-determinism of the system is resolved in advance by a *task-scheduler*, i.e. a sequence of tasks chosen in advance that trigger the actions among the enabled ones. They define the parallel composition for this type of automata, as well as the notion of implementation for TPIOA. Informally, the implementation of a Task-structured probabilistic Input/Output automata should look "similar" to the specification whatever will be the external environment of execution. Furthermore, they provide compositional results for the implementation relation. Even thought the formalism proposed in [5] (built on top of the one of [3]) has been already used in the formal proof of various cryptographic protocols [4, 21], this formalism does not capture the dynamicity of probabilistic dynamic systems such as peer-to-peer networks or blockchains systems where the set of participants dynamically changes or where subchains can be created or destroyed at run time [18].

Our contribution. In order to cope with dynamicity and probabilistic nature of modern distributed systems we propose an extension of the two formalisms introduced in [2] and [3]. Our extension uses a refined definition of probabilistic configuration automata in order to cope with dynamic actions. The main result of our formalism is as follows: the implementation of probabilistic configuration automata is monotonic to automata creation and destruction. That is, if systems $X_{\mathcal{A}}$ and $X_{\mathcal{B}}$ differ only in that $X_{\mathcal{A}}$ dynamically creates and destroys automaton $\mathcal{A}$ instead of creating and destroying automaton $\mathcal{B}$ as $X_{\mathcal{B}}$ does, and if $\mathcal{A}$ implements $\mathcal{B}$ (in the sense they cannot be distinguished by any external observer), then $X_{\mathcal{A}}$ implements $X_{\mathcal{B}}$. This result enables a design and refinement methodology based solely on the notion of externally visible behavior and permits the refinement of components and subsystems in isolation from the rest of the system. In our construction, we exhibit the need of considering only *creation-oblivious* schedulers in the implementation relation, i.e. a scheduler that, upon the (dynamic) creation of a sub-automaton $\mathcal{A}$, does not take into account the previous internal actions of $\mathcal{A}$ to output (randomly) a transition. Surprisingly, the task-schedulers introduced by Canetti & al. [3] are not creation-oblivious. Interestingly, an important contribution of the paper of independent interest is the proof technique we used in order to obtain our results. Differently from [2] and [3] which build their constructions mainly on induction techniques, we developed an elegant homomorphism based technique which aim to render the proofs modular. This proof technique can be easily adapted in order to further extend our framework with cryptography and time.

It should be noted that our work is an intermediate step before extending composable secure-emulation [5, 8] to dynamic settings. This extension is necessary for formal verification of secure dynamic distributed systems (e.g. blockchain systems).

Paper organization. The paper is organized as follows. Section 2 is dedicated to a brief introduction of the notion of probabilistic measure and recalls notations used in defining Signature I/O automata of [2]. Section 3 builds on the frameworks proposed in [2] and [3] in order to lay down the preliminaries of our formalism. More specifically, we introduce the definitions of probabilistic signed I/O automata and define their composition and implementation. In Section 4 we extend the definition of configuration automata proposed in [2] to probabilistic configuration automata then we define the composition of probabilistic configuration automata. Section 5 contains definitions related to the behavioural semantic of automata, e.g. executions, traces, etc. Section 6 introduces implementation relationship, which allows to formalise the idea that a concrete system is meeting the specification of an abstract object. The key result of our formalisation, the monotonicity of PSIOA implementations with respect to creation and destruction, is presented in Section 7 and demonstrated in the extended version. For a big picture, we recommend the reading of the warm up section of the extended version [7].

2 Preliminaries on probability and measure

We assume our reader is comfortable with basic notions of probability theory, such as σ-algebra and (discrete) probability measures. A measurable space is denoted by $(S, \mathcal{F}_S)$, where S is a set and $\mathcal{F}_S$ is a σ-algebra over S that is, $\mathcal{F}_S \subseteq \mathcal{P}(S)$, is closed under countable union and complementation and its members are called measurable sets ($\mathcal{P}(S)$ denotes the power set of S). The union of a collection $\{S_i\}_{i\in I}$ of pairwise disjoint sets indexed by a set I is written as $\biguplus_{i\in I} S_i$. A measure over $(S, \mathcal{F}_S)$ is a function $\eta : \mathcal{F}_s \to \mathbb{R}^{\geq 0}$, such that $\eta(\emptyset) = 0$ and for every countable collection of disjoint sets $\{S_i\}_{i\in I}$ in $\mathcal{F}_S$, $\eta(\biguplus_{i\in I} S_i) = \Sigma_{i\in I}\eta(S_i)$. A probability measure (resp. sub-probability measure) over $(S, \mathcal{F}_S)$ is a measure η such that $\eta(S) = 1$ (resp. $\eta(S) \leq 1$). A measure space is denoted by $(S, \mathcal{F}_S, \eta)$ where η is a measure on $(S, \mathcal{F}_S)$.

The product measure space $(S_1, \mathcal{F}_{s_1}, \eta_1) \otimes (S_2, \mathcal{F}_{s_2}, \eta_2)$ is the measure space $(S_1 \times S_2, \mathcal{F}_{s_1} \otimes \mathcal{F}_{s_2}, \eta_1 \otimes \eta_2)$, where $\mathcal{F}_{s_1} \otimes \mathcal{F}_{s_2}$ is the smallest σ-algebra generated by sets of the form $\{A \times B | A \in \mathcal{F}_{s_1}, B \in \mathcal{F}_{s_2}\}$ and $\eta_1 \otimes \eta_2$ is the unique measure s.t. for every $C_1 \in \mathcal{F}_{s_1}, C_2 \in \mathcal{F}_{s_2}$, $\eta_1 \otimes \eta_2(C_1 \times C_2) = \eta_1(C_1) \cdot \eta_2(C_2)$. If S is countable, we note $\mathcal{P}(S) = 2^S$. If S_1 and S_2 are countable, we have $2^{S_1} \otimes 2^{S_2} = 2^{S_1 \times S_2}$.

A discrete probability measure on a set S is a probability measure η on $(S, 2^S)$, such that, for each $C \subset S, \eta(C) = \sum_{c\in C} \eta(\{c\})$. We define $Disc(S)$ and $SubDisc(S)$ to be respectively, the set of discrete probability and sub-probability measures on S. In the sequel, we often omit the set notation when we denote the measure of a singleton set. For a discrete probability measure η on a set S, $supp(\eta)$ denotes the support of η, that is, the set of elements $s \in S$ such that $\eta(s) \neq 0$. Given set S and a subset $C \subset S$, the Dirac measure δ_C is the discrete probability measure on S that assigns probability 1 to C. For each element $s \in S$, we note δ_s for $\delta_{\{s\}}$.

If $\{m_i\}_{i\in I}$ is a countable family of measures on $(S, \mathcal{F}_S)$, and $\{p_i\}_{i\in I}$ is a family of non-negative values, then the expression $\sum_{i\in I} p_i m_i$ denotes a measure m on $(S, \mathcal{F}_S)$ such that, for each $C \in \mathcal{F}_S, m(C) = \sum_{i\in I} m_i f_i(C)$. A function $f : X \to Y$ is said to be measurable from $(X, \mathcal{F}_X) \to (Y, \mathcal{F}_Y)$ if the inverse image of each element of $\mathcal{F}_Y$ is an element of $\mathcal{F}_X$, that is, for each $C \in \mathcal{F}_Y$, $f^{-1}(C) \in \mathcal{F}_X$. In such a case, given a measure η on $(X, \mathcal{F}_X)$, the function $f(\eta)$ defined on $\mathcal{F}_Y$ by $f(\eta)(C) = \eta(f^{-1}(C))$ for each $C \in Y$ is a measure on $(Y, \mathcal{F}_Y)$ and is called the image measure of η under f.

Let $(Q_1, 2^{Q_1})$ and $(Q_2, 2^{Q_2})$ be two measurable sets. Let $(\eta_2, \eta_2) \in Disc(Q_1) \times Disc(Q_2)$. Let $f : Q_1 \to Q_2$. We note $\eta_1 \overset{f}{\leftrightarrow} \eta_2$ if the following is verified: (1) the restriction $\tilde{f}$ of f to $supp(\eta_1)$ is a bijection from $supp(\eta_1)$ to $supp(\eta_2)$ and (2) $\forall q \in supp(\eta)$, $\eta(q_1) = \eta_2(f(q_1))$.

3 Probabilistic Signature Input/Output Automata (PSIOA)

This section aims to introduce the first brick of our formalism: the probabilistic signature input/output automata (PSIOA). A PSIOA $\mathcal{A}$ is an automaton that can move from one *state* to another through *actions*. At each state q some actions can be triggered in its signature $sig(\mathcal{A})(q)$. Such an action leads to a new state with a certain probability. The fact that the signature can evolve throughout an execution is particularly convenient to model dynamicity.

3.1 PSIOA

We combine the SIOA of [2] with the PIOA of [19]. We use the signature approach from [2]. We assume the existence of a countable set *Autids* of unique probabilistic signature input/output automata (PSIOA) identifiers, an underlying universal set *Auts* of PSIOA, and a mapping $aut : Autids \rightarrow Auts$. $aut(\mathcal{A})$ is the PSIOA with identifier $\mathcal{A}$. We use "the automaton $\mathcal{A}$" to mean "the PSIOA with identifier $\mathcal{A}$". We use the letters $\mathcal{A}, \mathcal{B}$, possibly subscripted or primed, for PSIOA identifiers.

▶ **Definition 1** (PSIOA). *A PSIOA* $\mathcal{A} = (Q_{\mathcal{A}}, \bar{q}_{\mathcal{A}}, sig(\mathcal{A}), D_{\mathcal{A}})$*, where:*

- $Q_{\mathcal{A}}$ *is a countable set of* states*,* $(Q_{\mathcal{A}}, 2^{Q_{\mathcal{A}}})$ *is the* state space*,*
- $\bar{q}_{\mathcal{A}}$ *is the unique* start state*.*
- $sig(\mathcal{A}) : q \in Q_{\mathcal{A}} \mapsto sig(\mathcal{A})(q) = (in(\mathcal{A})(q), out(\mathcal{A})(q), int(\mathcal{A})(q))$ *is the signature function that maps each state to a triplet of mutually disjoint countable set of* actions*, respectively called* input*,* output *and* internal *actions.*
- $D_{\mathcal{A}} \subset Q_{\mathcal{A}} \times acts(\mathcal{A}) \times Disc(Q_{\mathcal{A}})$ *is the set of* probabilistic discrete transitions *where* $\forall (q, a, \eta) \in D_{\mathcal{A}} : a \in \widehat{sig}(\mathcal{A})(q)$*. If* (q, a, η) *is an element of* $D_{\mathcal{A}}$*, we write* $q \xrightarrow{a} \eta$ *and action* a *is said to be* enabled *at* q*. We note* $enabled(\mathcal{A}) : q \in Q_{\mathcal{A}} \mapsto enabled(\mathcal{A})(q)$ *where* $enabled(\mathcal{A})(q)$ *denotes the set of enabled actions at state* q*. We also note* $steps(\mathcal{A}) \triangleq \{(q, a, q') \in Q_{\mathcal{A}} \times acts(\mathcal{A}) \times Q_{\mathcal{A}} | \exists (q, a, \eta) \in D_{\mathcal{A}}, q' \in supp(\eta)\}$*.*

In addition $\mathcal{A}$ must satisfy the following conditions:

- $\mathbf{E_1}$ (input enabling) $\forall q \in Q_{\mathcal{A}}$, $in(\mathcal{A})(q) \subseteq enabled(\mathcal{A})(q)$.[1]
- $\mathbf{T_1}$ (Transition determinism): For every $q \in Q_{\mathcal{A}}$ and $a \in \widehat{sig}(\mathcal{A})(q)$ there is at most one $\eta_{(\mathcal{A},q,a)} \in Disc(Q_{\mathcal{A}})$, such that $(q, a, \eta_{(\mathcal{A},q,a)}) \in D_{\mathcal{A}}$.

We define $ext(\mathcal{A})(q)$, the external signature of $\mathcal{A}$ in state q, to be $ext(\mathcal{A})(q) = (in(\mathcal{A})(q), out(\mathcal{A})(q))$. We define $loc(\mathcal{A})(q)$, the local signature of $\mathcal{A}$ in state q, to be $loc(\mathcal{A})(q) = (out(\mathcal{A})(q), int(\mathcal{A})(q))$. For any signature component, generally, the $\widehat{\cdot}$ operator yields the union of sets of actions within the signature, e.g., $\widehat{sig}(\mathcal{A}) : q \in Q \mapsto \widehat{sig}(\mathcal{A})(q) = in(\mathcal{A})(q) \cup out(\mathcal{A})(q) \cup int(\mathcal{A})(q)$. Also we define $acts(\mathcal{A}) = \bigcup_{q \in Q} \widehat{sig}(\mathcal{A})(q)$, that is $acts(\mathcal{A})$ is the "universal" set of all actions that A could possibly trigger, in any state.

Later, we will define *execution fragments* as alternating sequences of states and actions with classic and natural consistency rules. But a subtlety will appear with the composability of set of automata at reachable states. Hence, we will define *execution fragments* after "local composability" and "probabilistic configuration automata".

[1] Since the signature is dynamic, we can require $\widehat{sig}(\mathcal{A}) = enabled(\mathcal{A})$

3.2 Local composition

The main aim of a formalism of concurrent systems is to compose several automata $\mathbf{A} = \{\mathcal{A}_1, ..., \mathcal{A}_n\}$ to capture the idea of an interaction between them and provide guarantees by composing the guarantees of the different elements of the system. Some syntactical rules have to be satisfied before defining the composition operation.

▶ **Definition 2** (Compatible signatures). *Let $S = \{sig_i\}_{i \in \mathcal{I}}$ be a set of signatures. Then S is compatible iff, $\forall i, j \in \mathcal{I}$, $i \neq j$, where $sig_i = (in_i, out_i, int_i)$, $sig_j = (in_j, out_j, int_j)$, we have: 1. $(in_i \cup out_i \cup int_i) \cap int_j = \emptyset$, and 2. $out_i \cap out_j = \emptyset$.*

▶ **Definition 3** (Composition of Signatures). *Let $\Sigma = (in, out, int)$ and $\Sigma' = (in', out', int')$ be compatible signatures. Then we define their composition $\Sigma \times \Sigma = (in \cup in' - (out \cup out'), out \cup out', int \cup int')$[2].*

Signature composition is clearly commutative and associative. Now we can define the compatibility of several automata at a state with the compatibility of their attached signatures. First we define compatibility at a state, and discrete transition for a set of automata for a particular compatible state.

▶ **Definition 4** (Compatibility at a state). *Let $\mathbf{A} = \{\mathcal{A}_1, ..., \mathcal{A}_n\}$ be a set of PSIOA. A* state *of $\mathbf{A}$ is an element $q = (q_1, ..., q_n) \in Q_{\mathbf{A}} \triangleq Q_{\mathcal{A}_1} \times ... \times Q_{\mathcal{A}_n}$. We note $q \upharpoonright \mathcal{A}_i \triangleq q_i$. We say $\mathcal{A}_1, ..., \mathcal{A}_n$ are (or $\mathbf{A}$ is)* compatible *at state q if $\{sig(\mathcal{A}_1)(q_1), ..., sig(\mathcal{A}_n)(q_n)\}$ is a set of compatible signatures. In this case we note $sig(\mathbf{A})(q) \triangleq sig(\mathcal{A}_1)(q_1) \times ... \times sig(\mathcal{A}_n)(q_n)$ as per definition 3 and we note $\eta_{(\mathbf{A},q,a)} \in Disc(Q_{\mathbf{A}})$, s.t. $\forall a \in \widehat{sig}(\mathbf{A})(q)$, $\eta_{(\mathbf{A},q,a)} = \eta_1 \otimes ... \otimes \eta_n$ where $\forall j \in [1, n]$, $\eta_j = \eta_{(\mathcal{A}_j, q_j, a)}$ if $a \in sig(\mathcal{A}_j)(q_j)$ and $\eta_j = \delta_{q_j}$ otherwise. Moreover, we note $steps(\mathbf{A}) = \{(q, a, q') | q, q' \in Q_{\mathbf{A}}, a \in sig(\mathbf{A})(q), q' \in supp(\eta_{(\mathbf{A},q,a)})\}$. Finally, we note $\bar{q}_{\mathbf{A}} = (\bar{q}_{\mathcal{A}_1}, ..., \bar{q}_{\mathcal{A}_n})$.*

Let us note that an action a shared by two automata becomes an output action and not an internal action after composition. First, it permits the possibility of further communication using a. Second, it allows associativity. If this property is counter-intuitive, it is always possible to use the classic hiding operator that "hides" the output actions transforming them into internal actions.

▶ **Definition 5** (Hiding operator). *Let $sig = (in, out, int)$ be a signature and H a set of actions. We note $hide(sig, H) \triangleq (in, out \setminus H, int \cup (out \cap H))$.*

Let $\mathcal{A} = (Q_{\mathcal{A}}, \bar{q}_{\mathcal{A}}, sig(\mathcal{A}), D_{\mathcal{A}})$ be a PSIOA. Let $h : q \in Q_{\mathcal{A}} \mapsto h(q) \subseteq out(\mathcal{A})(q)$. We note $hide(\mathcal{A}, h) \triangleq (Q_{\mathcal{A}}, \bar{q}_{\mathcal{A}}, sig'(\mathcal{A}), D_{\mathcal{A}})$, where $sig'(\mathcal{A}) : q \in Q_{\mathcal{A}} \mapsto hide(sig(\mathcal{A})(q), h(q))$. Clearly, $hide(\mathcal{A}, h)$ is a PSIOA.

4 Probabilistic Configuration Automata

We combine the notion of configuration of [2] with the probabilistic setting of [19]. A configuration is a set of automata attached with their current states. This will be a very useful tool to define dynamicity by mapping the state of an automaton of a certain "layer" to a configuration of automata of lower layer, where the set of automata in the configuration can dynamically change from on state of the automaton of the upper level to another one.

[2] not to be confused with Cartesian product. We keep this notation to stay as close as possible to the literature.

4.1 Configuration

▶ **Definition 6** (Configuration). *A configuration is a pair* $(\mathbf{A}, \mathbf{S})$ *where*

- $\mathbf{A} = \{\mathcal{A}_1, ..., \mathcal{A}_n\}$ *is a finite set of PSIOA identifiers and*
- $\mathbf{S}$ *maps each* $\mathcal{A}_k \in \mathbf{A}$ *to a state of* $\mathcal{A}_k$.

In distributed computing, configuration usually refers to the union of states of ***all*** *the automata of the "system". Here, there is a subtlety, since it captures a set of some automata* ($\mathbf{A}$) *in their current state* ($\mathbf{S}$)*, but the set of automata of the systems will not be fixed in the time.*

Since, (1) $\{\mathbf{A} \in \mathcal{P}(Autids) | \mathbf{A} \text{ is finite}\}$ *is countable, (2)* $\forall \mathcal{A} \in Autids, Q_{\mathcal{A}}$ *is countable by definition 1 of PSIOA and (3) the cartesian product of countable sets is a countable set, we can deduce that the set* Q_{conf} *of configurations is countable.*

▶ **Definition 7** (Compatible configuration). *A configuration* $(\mathbf{A}, \mathbf{S})$*, with* $\mathbf{A} = \{\mathcal{A}_1, ..., \mathcal{A}_n\}$*, is compatible iff the set* $\mathbf{A}$ *is compatible at state* $(\mathbf{S}(\mathcal{A}_1), ..., \mathbf{S}(\mathcal{A}_n))$ *as per definition 4.*

▶ **Definition 8** (Intrinsic attributes of a configuration). *Let* $C = (\mathbf{A}, \mathbf{S})$ *be a compatible configuration. Then we define*

- $auts(C) = \mathbf{A}$ *represents the automata of the configuration,*
- $map(C) = \mathbf{S}$ *maps each automaton of the configuration with its current state,*
- $TS(C) = (\mathbf{S}(\mathcal{A}_1), ..., \mathbf{S}(\mathcal{A}_n))$ *yields the tuple of states of the automata of the configuration.*
- $sig(C) = (in(C), out(C), int(C)) = sig(auts(C), TS(C))$ *in the sense of definition 4, is called the intrinsic signature of the configuration*

Here we define a reduced configuration as a configuration deprived of the automata that are in the very particular state where their current signatures are the empty set. This mechanism will be used later to capture the idea of destruction of an automaton.

▶ **Definition 9** (Reduced configuration). $reduce(C) = (\mathbf{A}', \mathbf{S}')$*, where* $\mathbf{A}' = \{\mathcal{A} | \mathcal{A} \in \mathbf{A} \text{ and } sig(\mathcal{A})(\mathbf{S}(\mathcal{A})) \neq \emptyset\}$ *and* $\mathbf{S}'$ *is the restriction of* $\mathbf{S}$ *to* $\mathbf{A}'$*, noted* $\mathbf{S} \restriction \mathbf{A}'$ *in the remaining.*

A configuration C *is a reduced configuration iff* $C = reduce(C)$.

We will define some probabilistic transition from configurations to others where some automata can be destroyed or created. To define it properly, we start by defining "preserving transition" where no automaton is neither created nor destroyed and then we define above this definition the notion of configuration transition.

▶ **Definition 10** (From preserving distribution to intrinsic transition).

- *(preserving distribution) Let* $\eta_p \in Disc(Q_{conf})$. *We say* η_p *is a* preserving distribution *if it exists a finite set of automata* $\mathbf{A}$*, called* family support of η_p*, s.t.* $\forall (\mathbf{A}', \mathbf{S}') \in supp(\eta_p), \mathbf{A} = \mathbf{A}'$.
- *(preserving configuration transition* $C \stackrel{a}{\rightharpoonup} \eta_p$*) Let* $C = (\mathbf{A}, \mathbf{S})$ *be a compatible configuration,* $a \in \widehat{sig}(C)$. *Let* η_p *be the unique preserving distribution of* $Disc(Q_{conf})$ *such that (1) the family support of* η_p *is* $\mathbf{A}$ *and (2)* $\eta_p \stackrel{TS}{\leftrightarrow} \eta_{(\mathbf{A}, TS(C), a)}$. *We say that* (C, a, η_p) *is a* preserving configuration transition*, noted* $C \stackrel{a}{\rightharpoonup} \eta_p$.
- *(*$\eta_p \uparrow \varphi$*) Let* $\eta_p \in Disc(Q_{conf})$ *be a preserving distribution with* $\mathbf{A}$ *as family support. Let* φ *be a finite set of of PSIOA identifiers with* $\mathbf{A} \cap \varphi = \emptyset$. *Let* $C_\varphi = (\varphi, S_\varphi) \in Q_{conf}$ *with* $\forall \mathcal{A}_j \in \varphi, S_\varphi(\mathcal{A}_j) = \bar{q}_{\mathcal{A}_j}$. *We note* $\eta_p \uparrow \varphi$ *the unique element of* $Disc(Q_{conf})$ *verifying* $\eta_p \stackrel{u}{\leftrightarrow} (\eta_p \uparrow \varphi)$ *with* $u : C \in supp(\eta_p) \mapsto (C \cup C_\varphi)$.

- *(distribution reduction) Let $\eta \in Disc(Q_{conf})$. We note $reduce(\eta)$ the element of $Disc(Q_{conf})$ verifying $\forall c \in Q_{conf}$, $(reduce(\eta))(c) = \Sigma_{(c' \in supp(\eta), c = reduce(c'))} \eta(c')$*
- *(intrinsic transition $C \stackrel{a}{\Longrightarrow}_{\varphi} \eta$) Let $C = (\mathbf{A}, \mathbf{S})$ be a compatible configuration, let $a \in \widehat{sig}(C)$, let φ be a finite set of of PSIOA identifiers with $\mathbf{A} \cap \varphi = \emptyset$. We note $C \stackrel{a}{\Longrightarrow}_{\varphi} \eta$, if $\eta = reduce(\eta_p \uparrow \varphi)$ with $C \stackrel{a}{\rightharpoonup} \eta_p$. In this case, we say that η is* generated by *η_p* and *φ.*

Preserving configuration transition (C, a, η_p) is the intuitive transition for configurations, corresponding to the transition $(TS(C), a, \eta_{(auts(C), TS(C), a)})$. The operator $\uparrow \varphi$ describes the deterministic creation of automata in φ, who will be appear at their respective start states. The *reduce* operator enables to remove "destroyed" automata from the possibly returned configurations (see Figure 1).

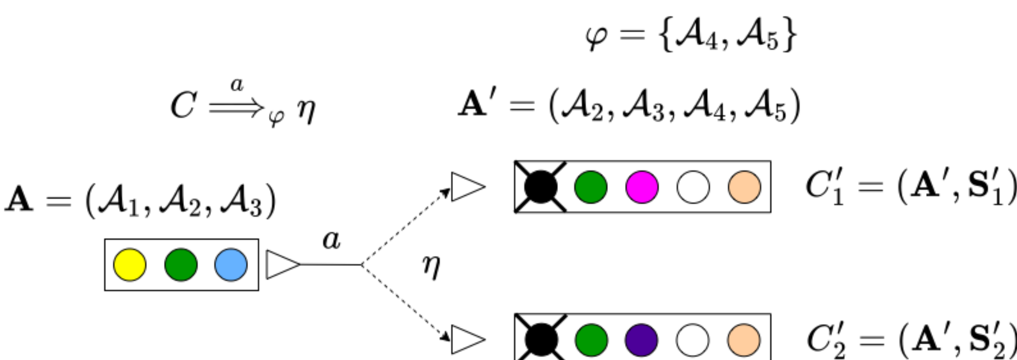


Figure 1 An intrinsic transition where $\mathcal{A}_1$ is destroyed deterministically and automata in $\varphi = \{\mathcal{A}_4, \mathcal{A}_5\}$ are created deterministically. First, we have the preserving disribution η_p s.t. $C \stackrel{a}{\rightharpoonup} \eta_p$ with $\eta_p \stackrel{TS}{\leftrightarrow} \eta_{(\mathbf{A}, TS(C), a)}$. Second, we take into account the created automata in φ, captured by the distribution $\eta_p \uparrow \varphi$. Third, we remove the automata in a particular state with associated empty signature ($\mathcal{A}_1$ in our example). This is captured by distribution $reduce(\eta_p \uparrow \varphi)$.

4.2 Probabilistic configuration automata (PCA)

Now we are ready to define our probabilistic configuration automata (see figure 2). Such an automaton define a strong link with a dynamic configuration.

▶ **Definition 11** (Probabilistic Configuration Automaton). *A probabilistic configuration automaton (PCA) X consists of the following components:*

1. *A probabilistic signature I/O automaton $psioa(X)$. For brevity, we define $Q_X = Q_{psioa(X)}, \bar{q}_X = \bar{q}_{psioa(X)}$, $sig(X) = sig(psioa(X)), steps(X) = steps(psioa(X))$, and likewise for all other (sub)components and attributes of $psioa(X)$.*
2. *A configuration mapping $config(X)$ with domain Q_X and such that, for all $q \in Q_X$, $config(X)(q)$ is a reduced compatible configuration.*
3. *For each $q \in Q_X$, a mapping $created(X)(q)$ with domain $sig(X)(q)$ and such that $\forall a \in sig(X)(q)$, $created(X)(q)(a) \subseteq Autids$ with $created(X)(q)(a)$ finite.*
4. *A hidden-actions mapping $hidden\text{-}actions(X)$ with domain Q_X and such that $hidden\text{-}actions(X)(q) \subseteq out(config(X)(q))$.*

and satisfies the following constraints, for every $q \in Q_X$, $C = config(X)(q)$, $H = hidden\text{-}actions(q)$.

1. *(start states preservation) If* $config(X)(\bar{q}_X) = (\mathbf{A}, \mathbf{S})$*, then* $\forall \mathcal{A}_i \in \mathbf{A}, \mathbf{S}(\mathcal{A}_i) = \bar{q}_{\mathcal{A}_i}$.
2. *(top/down transition preservation) If* $(q, a, \eta_{(X,q,a)}) \in D_X$*, then* $\exists \eta' \in Disc(Q_{conf})$ *s.t.* $\eta_{(X,q,a)} \overset{c}{\leftrightarrow} \eta'$ *with* $C \overset{a}{\Longrightarrow}_{\varphi} \eta'$*, where* $\varphi = created(X)(q)(a)$ *and* $c = config(X)$.
3. *(bottom/up transition preservation) If* $q \in Q_X$ *and* $C \overset{a}{\Longrightarrow}_{\varphi} \eta'$ *for some action* a*,* $\varphi = created(X)(q)(a)$*, and reduced compatible probabilistic measure* $\eta' \in Disc(Q_{conf})$*, then* $(q, a, \eta_{(X,q,a)}) \in D_X$*, and* $\eta_{(X,q,a)} \overset{c}{\leftrightarrow} \eta'$ *where* $c = config(X)$.
4. *(signature preservation modulo hiding)* $\forall q \in Q_X$ *,* $sig(X)(q) = hide(sig(C), H)$.

This definition, proposed in a deterministic fashion in [2], captures dynamicity of the system. Each state is linked with a configuration. The set of automata of the configuration can change during an execution. A sub-automaton $\mathcal{A}$ is created from state q by the action a if $\mathcal{A} \in created(X)(q)(a)$. A sub-automaton $\mathcal{A}$ is destroyed if the non-reduced attached configuration distribution leads to a configuration where $\mathcal{A}$ is in a state $q^{\phi}_{\mathcal{A}}$ s.t. $\widehat{sig}(\mathcal{A})(q^{\phi}_{\mathcal{A}}) = \emptyset$. Then the corresponding reduced configuration will not hold $\mathcal{A}$. The last constraint states that the signature of a state q of X must be the same as the signature of its corresponding configuration $config(X)(q)$, except for the possible effects of hiding operators, so that some outputs of $config(X)(q)$ may be internal actions of X in state q.

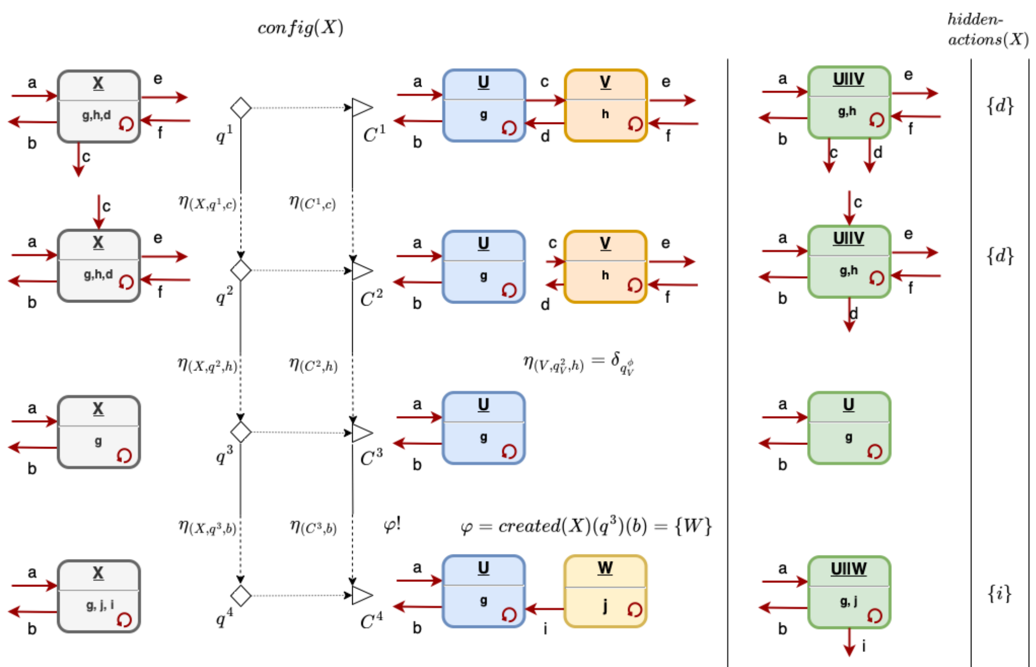


Figure 2 A PCA life cycle. V is destroyed at step (q^2, h, q^3), while W is created at step (q^3, b, q^4).

As for PSIOA, we can define hiding operator applied to PCA.

▶ **Definition 12** (Hiding on PCA)**.** *Let* X *be a PCA. Let* $h : q \in Q_X \mapsto h(q) \subseteq out(X)(q)$*. We note* $hide(X, h)$ *the PCA* X' *that differs from* X *only on*

- $psioa(X') = hide(psioa(X), h)$
- $sig(X') = hide(sig(X), h)$ *and*
- $\forall q \in Q_X = Q_{X'}$*,* $hidden\text{-}actions(X')(q) = hidden\text{-}actions(X)(q) \cup h(q)$.

The notion of local compatibility can be naturally extended to set of PCA.

▶ **Definition 13** (PCA compatible at a state). *Let $\mathbf{X} = \{X_1, ..., X_n\}$ be a set of PCA. Let $q = (q_1, ..., q_n) \in Q_{X_1} \times ... \times Q_{X_n}$. Let us note $C_i = (\mathbf{A}_i, \mathbf{S}_i) = config(X_i)(q_i)$, $\forall i \in [1, n]$. The PCA in $\mathbf{X}$ are compatible at state q iff*[3]*:*

1. *PSIOA compatibility: $psioa(X_1), ..., psioa(X_n)$ are compatible at $q_{\mathbf{X}}$.*
2. *Sub-automaton exclusivity: $\forall i, j \in [1 : n], i \neq j : \mathbf{A}_i \cap \mathbf{A}_j = \emptyset$.*
3. *Creation exclusivity: $\forall i, j \in [1 : n], i \neq j, \forall a \in \widehat{sig}(X_i)(q_i) \cap \widehat{sig}(X_j)(q_j)$: $created(X_i)(q_i)(a) \cap created(X_j)(q_j)(a) = \emptyset$.*

If $\mathbf{X}$ is compatible at state q, for every action $a \in \widehat{sig}(psioa(\mathbf{X}))(q)$, we note $\eta_{(\mathbf{X},q,a)} = \eta_{(psioa(\mathbf{X}),q,a)}$ and we extend this notation with $\eta_{(\mathbf{X},q,a)} = \delta_q$ if $a \notin \widehat{sig}(psioa(\mathbf{X}))(q)$.

5 Executions, reachable states, partially-compatible automata

In previous sections, we have described how to model probabilistic transitions that might lead to the creation and destruction of some components of the system. In this section, we will define pseudo execution fragments of a set of automata to model the run of a set $\mathbf{A}$ of several dynamic systems interacting with each others. With such a definition, we will kill two birds with one stone, since it will allow to define *reachable states* of $\mathbf{A}$ and then compatibility of $\mathbf{A}$ as compatibility of $\mathbf{A}$ at each reachable state.

5.1 Executions, reachable states, traces

▶ **Definition 14** (Pseudo execution, reachable states, partial-compatibility). *Let $\mathbf{A} = \{\mathcal{A}_1, ..., \mathcal{A}_n\}$ be a finite set of PSIOA (resp. PCA). A* pseudo execution fragment *of $\mathbf{A}$ is a finite or infinite sequence $\alpha = q^0 a^1 q^1 a^2 ...$ of alternating states and actions, such that:*

1. *If α is finite, it ends with a state. In that case, we note $lstate(\alpha)$ the last state of α.*
2. *$\mathbf{A}$ is compatible at each state of α, with the potential exception of $lstate(\alpha)$ if α is finite.*
3. *for ever action a^i, $(q^{i-1}, a^i, q^i) \in steps(\mathbf{A})$.*

The first state of a pseudo execution fragment α is noted $fstate(\alpha)$. A pseudo execution fragment α of $\mathbf{A}$ is a pseudo execution of $\mathbf{A}$ if $fstate(\alpha) = \bar{q}_{\mathbf{A}}$. The length $|\alpha|$ of a finite pseudo execution fragment α is the number of actions in α. A state q of $\mathbf{A}$ is said reachable *if there is a pseudo execution α s.t. $lstate(\alpha) = q$. We note $Reachable(\mathbf{A})$ the set of* reachable *states of $\mathbf{A}$. If $\mathbf{A}$ is compatible at every reachable state q, $\mathbf{A}$ is said* partially-compatible.

▶ **Definition 15** (Executions, concatenations). *Let $\mathcal{A}$ be an automaton. An* execution fragment *(resp.* execution*) of $\mathcal{A}$ is a pseudo execution fragment (resp. pseudo execution) of $\{\mathcal{A}\}$. We use $Frags(\mathcal{A})$ (resp., $Frags^*(\mathcal{A})$) to denote the set of all (resp., all finite) execution fragments of $\mathcal{A}$. $Execs(\mathcal{A})$ (resp. $Execs^*(\mathcal{A})$) denotes the set of all (resp., all finite) executions of $\mathcal{A}$.*

We define a concatenation operator $\frown$ for execution fragments as follows:
If $\alpha = q^0 a^1 q^1 ... a^n q^n \in Frags^*(\mathcal{A})$ and $\alpha' = q^{0\prime} a^{1\prime} q^{1\prime} ... \in Frags^*(\mathcal{A})$, we define $\alpha \frown \alpha' \triangleq q^0 a^1 q^1 ... a^n q^n \; a^{1\prime} q^{1\prime} ...$ only if $s^0 = q^n$, otherwise $\alpha \frown \alpha'$ is undefined. Hence the notation $\alpha \frown \alpha'$ implicitly means $fstate(\alpha') = lstate(\alpha)$. Let $\alpha, \alpha' \in Frags(\mathcal{A})$, then α is a prefix of α', noted $\alpha \leq \alpha'$, iff $\exists \alpha'' \in Frags(\mathcal{A})$ such that $\alpha' = \alpha \frown \alpha''$.

The *trace* of an execution α represents its externally visible part, i.e. the external actions.

[3] We can remark that the conjunction of PSIOA compatibility and sub-automata exclusivity implies the compatibility of respective configurations as defined later in definition 19

▶ **Definition 16** (Traces). *Let $\mathcal{A}$ be a PSIOA (resp. PCA). Let $q^0 \in Q_{\mathcal{A}}$, $(q, a, q') \in steps(\mathcal{A})$, $\alpha, \alpha' \in Execs^*(\mathcal{A}) \times Execs(\mathcal{A})$ with $fstate(\alpha') = lstate(\alpha)$.*

$trace_{\mathcal{A}}(q^0)$ *is the empty sequence, noted* λ,

$$trace_{\mathcal{A}}(qaq') \begin{cases} a \text{ if } a \in \widehat{ext}(\mathcal{A})(q) \\ \lambda \text{ otherwise.} \end{cases},$$

$trace_{\mathcal{A}}(\alpha^\frown\alpha') = trace_{\mathcal{A}}(\alpha)^\frown trace_{\mathcal{A}}(\alpha')$

We say that β is a trace of $\mathcal{A}$ if $\exists \alpha \in Execs(\mathcal{A})$ with $\beta = trace_{\mathcal{A}}(\alpha)$. We note $Traces(\mathcal{A})$ (resp. $Traces^*(\mathcal{A})$, resp. $Traces^\omega(\mathcal{A})$) the set of traces (resp. finite traces, resp. infinite traces) of $\mathcal{A}$. When the automaton $\mathcal{A}$ is understood from context, we write simply $trace(\alpha)$.

The projection of a pseudo-execution α on an automaton $\mathcal{A}_i$, noted $\alpha \upharpoonright \mathcal{A}_i$, represents the contribution of $\mathcal{A}_i$ to this execution.

▶ **Definition 17** (Projection). *Let $\mathbf{A}$ be a set of PSIOA (resp. PCA), let $\mathcal{A}_i \in \mathbf{A}$. We define projection operator $\upharpoonright$ recursively as follows: For every $(q, a, q') \in steps(\mathbf{A})$, for every α, α' being two pseudo executions of $\mathbf{A}$ with $fstate(\alpha') = lstate(\alpha)$.*

$$(q, a, q') \upharpoonright \mathcal{A}_i = \begin{cases} (q \upharpoonright \mathcal{A}_i), a, (q' \upharpoonright \mathcal{A}_i) \text{ if } a \in \widehat{sig}(\mathcal{A}_i)(q \upharpoonright \mathcal{A}_i) \\ (q \upharpoonright \mathcal{A}_i) = (q' \upharpoonright \mathcal{A}_i) \text{ otherwise.} \end{cases},$$

$(\alpha^\frown\alpha') \upharpoonright \mathcal{A}_i = (\alpha \upharpoonright \mathcal{A}_i)^\frown(\alpha' \upharpoonright \mathcal{A}_i)$

5.2 PSIOA and PCA composition

We are ready to define composition operator, the most important operator for concurrent systems.

▶ **Definition 18** (PSIOA partial-composition). *If $\mathbf{A} = \{\mathcal{A}_1, ..., \mathcal{A}_n\}$ is a partially-compatible set of PSIOA, with $\mathcal{A}_i = (Q_{\mathcal{A}_i}, \bar{q}_{\mathcal{A}_i}, sig(\mathcal{A}_i), D_{\mathcal{A}_i})$, then their partial-composition $\mathcal{A}_1||...||\mathcal{A}_n$, is defined to be $\mathcal{A} = (Q_{\mathcal{A}}, \bar{q}_{\mathcal{A}}, sig(\mathcal{A}), D_{\mathcal{A}})$, where:*

- $Q_{\mathcal{A}} = Reachable(\mathbf{A})$
- $\bar{q}_{\mathcal{A}} = (\bar{q}_{\mathcal{A}_1}, ..., \bar{q}_{\mathcal{A}_n})$
- $sig(\mathcal{A}) : q \in Q_{\mathcal{A}} \mapsto sig(\mathcal{A})(q) = sig(\mathbf{A})(q)$
- $D_{\mathcal{A}} = \{(q, a, \eta_{(\mathbf{A},q,a)}) | q \in Q_{\mathcal{A}}, a \in \widehat{sig}(\mathbf{A})(q)\}$

▶ **Definition 19** (Union of configurations). *Let $C_1 = (\mathbf{A}_1, \mathbf{S}_1)$ and $C_2 = (\mathbf{A}_2, \mathbf{S}_2)$ be configurations such that $\mathbf{A}_1 \cap \mathbf{A}_2 = \emptyset$. Then, the union of C_1 and C_2, denoted $C_1 \cup C_2$, is the configuration $C = (\mathbf{A}, \mathbf{S})$ where $\mathbf{A} = \mathbf{A}_1 \cup \mathbf{A}_2$ and $\mathbf{S}$ agrees with $\mathbf{S}_1$ on $\mathbf{A}_1$, and with $\mathbf{S}_2$ on $\mathbf{A}_2$. Moreover, if $C_1 \cup C_2$ is a compatible configuration, we say that C_1 and C_2 are* compatible configurations. *It is clear that configuration union is commutative and associative. Hence, we will freely use the n-ary notation $C_1 \cup ... \cup C_n$, whenever $\forall i, j \in [1 : n], i \neq j, auts(C_i) \cap auts(C_j) = \emptyset$.*

▶ **Definition 20** (PCA partial-composition). *If $\mathbf{X} = \{X_1, ..., X_n\}$ is a partially-compatible set of PCA, then their partial-composition $X_1||...||X_n$, is defined to be the automaton X, with the same components than a PCA, s.t. $psioa(X) = psioa(X_1)||...||psioa(X_n)$ and $\forall q \in Q_X$:*

- $config(X)(q) = \bigcup_{i\in[1,n]} config(X_i)(q \upharpoonright X_i)$
- $\forall a \in \widehat{sig}(X)(q)$, $created(X)(q)(a) = \bigcup_{i\in[1,n]} created(X_i)(q \upharpoonright X_i)(a)$, *with the convention* $created(X_i)(q_i)(a) = \emptyset$ *if* $a \notin \widehat{sig}(X_i)(q_i)$
- $hidden\text{-}actions(q) = \bigcup_{i\in[1,n]} hidden\text{-}actions(X_i)(q \upharpoonright X_i)$

▶ **Theorem 21** (PCA closeness under composition). *Let $X_1, ..., X_n$, be partially-compatible PCA. Then $X = X_1||...||X_n$ is a PCA.*

6 Scheduler, measure on executions, implementation

An inherent non-determinism appears for concurrent systems. Indeed, after composition (or even before), it is natural to obtain a state with several enabled actions. The most common case is the reception of two concurrent messages in flight from two different processes. This non-determinism must be solved if we want to define a probability measure on the automata executions and be able to say that a situation is likely to occur or not. To solve the non-determinism, we use a scheduler that chooses an enabled action from a signature.

6.1 General definition and probabilistic space on execution fragments

A scheduler is hence a function that takes an execution fragment as input and outputs the probability distribution on the set of transitions that will be triggered. We reuse the formalism from [19] with the syntax from [3].

▶ **Definition 22** (Scheduler). *A* scheduler *of a PSIOA (resp. PCA) $\mathcal{A}$ is a function*

$$\sigma : Frags^*(\mathcal{A}) \rightarrow SubDisc(D_{\mathcal{A}}) \text{ such that } (q, a, \eta) \in supp(\sigma(\alpha)) \text{ implies } q = lstate(\alpha).$$

Here $SubDisc(D_{\mathcal{A}})$ is the set of discrete sub-probability distributions on $D_{\mathcal{A}}$. Loosely speaking, σ decides (probabilistically) which transition to take after each finite execution fragment α. Since this decision is a discrete sub-probability measure, it may be the case that σ chooses to halt after α with non-zero probability: $1 - \sigma(\alpha)(D_{\mathcal{A}}) > 0$. We note $schedulers(\mathcal{A})$ the set of schedulers of $\mathcal{A}$.

▶ **Definition 23** (Measure $\epsilon_{\sigma,\alpha}$ generated by a scheduler and a fragment). *A scheduler σ and a finite execution fragment α generate a measure $\epsilon_{\sigma,\alpha}$ on the sigma-algebra $\mathcal{F}_{Frags(\mathcal{A})}$ generated by cones of execution fragments, where each cone $C_{\alpha'}$ is the set of execution fragments that have α' as a prefix, i.e. $C_{\alpha'} = \{\alpha \in Frags(\mathcal{A}) | \alpha' \leq \alpha\}$. The measure of a cone $C_{\alpha'}$ is defined recursively as follows:*

$$\epsilon_{\sigma,\alpha}(C_{\alpha'}) =: \begin{cases} 0 & \text{if both } \alpha' \nleq \alpha \text{ and } \alpha \nleq \alpha' \\ 1 & \text{if } \alpha' \leq \alpha \\ \epsilon_{\sigma,\alpha}(C_{\alpha''}) \cdot \sigma(\alpha'')(\eta_{(\mathcal{A},q',a)}) \cdot \eta_{(\mathcal{A},q',a)}(q) & \text{if } \alpha \leq \alpha'' \text{ and } \alpha' = \alpha'' {}^\frown q'aq \end{cases}$$

In the remaining part of the paper, we will mainly focus on probabilistic executions of $\mathcal{A}$ of the form $\epsilon_\sigma \triangleq \epsilon_{\sigma,\delta_{\bar{q}_{\mathcal{A}}}} = \epsilon_{\sigma,\bar{q}_{\mathcal{A}}}$. Hence, we will deal with probablistic space of the form $(Execs(\mathcal{A}), \mathcal{F}_{Execs(\mathcal{A})}, \epsilon_\sigma)$.

Scheduler Schema

Without restriction, a scheduler could become a too powerful adversary for practical applications. Hence, it is common to only consider a subset of schedulers, called a *scheduler schema*. Typically, a classic limitation is often described by a scheduler with "partial online information". Some formalism has already been proposed in [19] (section 5.6) to impose the scheduler that its choices are correlated for executions fragments in the same equivalence class where both the equivalence relation and the correlation must to be defined. This idea has been reused and simplified in [4] that defines equivalence classes on actions, called *tasks*. Then, a task-scheduler (a.k.a. "off-line" scheduler) selects a sequence of tasks $T_1, T_2, ...$ in advance that it cannot modify during the execution of the automaton. After each transition, the next task T_i triggers an enabled action if there is no ambiguity and is ignored otherwise. One of our main contribution, the theorem of implementation monotonicity w.r.t. PSIOA creation, is ensured only for a certain scheduler schema, so-called *creation-oblivious*. However, we will see that the practical set of task-schedulers are not creation-oblivious.

▶ **Definition 24** (Scheduler schema). *A* scheduler schema *is a function that maps every PSIOA (resp. PCA) $\mathcal{A}$ to a subset of $schedulers(\mathcal{A})$.*

6.2 Implementation

In last subsection, we defined a measure of probability on executions with the help of a scheduler to solve non-determinism. Now we can define the notion of implementation. The intuition behind this notion is the fact that any environment $\mathcal{E}$ that would interact with both $\mathcal{A}$ and $\mathcal{B}$, would not be able to distinguish $\mathcal{A}$ from $\mathcal{B}$. The classic use-case is to formally show that a (potentially very sophisticated) algorithm implements a specification.

For us, an environment is simply a partially-compatible automaton, but in practice, he will play the role of a "distinguisher".

▶ **Definition 25** (Environment). *A probabilistic environment for PSIOA $\mathcal{A}$ is a PSIOA $\mathcal{E}$ such that $\mathcal{A}$ and $\mathcal{E}$ are partially-compatible. We note $env(\mathcal{A})$ the set of environments of $\mathcal{A}$.*

Now we define *perception function* which is a function that captures the pieces of information that could be obtained by an external observer to attempt a distinction.

▶ **Definition 26** (Perception function). *A* perception-function *is a function $f_{(.,.)}$ parametrized by a pair $(\mathcal{E}, \mathcal{A})$ of PSIOA (resp. PCA) where $\mathcal{E} \in env(\mathcal{A})$ s.t.*

- *(Measurability) For every pair $(\mathcal{E}, \mathcal{A})$ of PSIOA (resp. PCA) where $\mathcal{E} \in env(\mathcal{A})$, $f_{(\mathcal{E},\mathcal{A})}$ is a measurable function from $(Execs(\mathcal{E}||\mathcal{A}), \mathcal{F}_{Execs(\mathcal{E}||\mathcal{A})})$ to some measurable space $(G_{(\mathcal{E},\mathcal{A})}, \mathcal{F}_{G_{(\mathcal{E},\mathcal{A})}})$ that has to be made explicit.*
- *(Stability by composition) For every quadruplet of PSIOA $(\mathcal{A}_1, \mathcal{A}_2, \mathcal{B}, \mathcal{E})$, s.t. $\mathcal{B}$ is partially compatible with $\mathcal{A}_1$ and $\mathcal{A}_2$, $\mathcal{E} \in env(\mathcal{B}||\mathcal{A}_1) \cap env(\mathcal{B}||\mathcal{A}_2)$, $\forall (C_1, C_2) \in \mathcal{F}_{Execs(\mathcal{E}||\mathcal{B}||\mathcal{A}_1)} \times \mathcal{F}_{Execs(\mathcal{E}||\mathcal{B}||\mathcal{A}_2)}$, $f_{(\mathcal{E}||\mathcal{B},\mathcal{A}_1)}(C_1) = f_{(\mathcal{E}||\mathcal{B},\mathcal{A}_2)}(C_2) \Longrightarrow f_{(\mathcal{E},\mathcal{B}||\mathcal{A}_1)}(C_1) = f_{(\mathcal{E},\mathcal{B}||\mathcal{A}_2)}(C_2)$.*

The first property is a standard measurability requirement, while the second captures the fact that an environment $\mathcal{E}$ does not have a greater power of distinction than $\mathcal{E}$ composed with another system $\mathcal{B}$. Any reasonable function that captures the perception of an automaton $\mathcal{A}$ by an environment $\mathcal{E} \in env(\mathcal{A})$ should be a perception function.

▶ **Lemma 27.** *The function $trace_{(.,.)}$ and $proj_{(.,.)}$ s.t. for every PSIOA (resp. PCA) $\mathcal{A}$, $\forall \mathcal{E} \in env(\mathcal{A})$, $trace_{(\mathcal{E},\mathcal{A})} : \alpha \in Execs(\mathcal{E}||\mathcal{A}) \mapsto trace_{(\mathcal{E}||\mathcal{A})}(\alpha)$ and $proj_{(\mathcal{E},\mathcal{A})} : \alpha \in Execs(\mathcal{E}||\mathcal{A}) \mapsto \alpha \upharpoonright \mathcal{E}$ are perception functions.*

Since a perception-function $f_{(.,.)}$ is measurable, we can define the image measure of $\epsilon_{\sigma,\mu}$ under $f_{(\mathcal{E},\mathcal{A})}$, i.e. the probability to obtain a certain external perception under a certain scheduler σ and a certain probability distribution μ on the starting executions.

▶ **Definition 28** (f-*dist*). *Let $f_{(.,.)}$ be a perception-function. Let $(\mathcal{E}, \mathcal{A})$ be a pair of PSIOA where $\mathcal{E} \in env(\mathcal{A})$. Let μ be a probability measure on $(Execs(\mathcal{E}||\mathcal{A}), \mathcal{F}_{Execs(\mathcal{E}||\mathcal{A})})$, and $\sigma \in schedulers(\mathcal{E}||\mathcal{A})$. We define $f\text{-}dist_{(\mathcal{E},\mathcal{A})}(\sigma, \mu)$, to be the image measure of $\epsilon_{\sigma,\mu}$ under $f_{(\mathcal{E},\mathcal{A})}$ (i.e. the function that maps any $C \in \mathcal{F}_{G_{(\mathcal{E},\mathcal{A})}}$ to $\epsilon_{\sigma,\mu}(f^{-1}_{(\mathcal{E},\mathcal{A})}(C))$) . We note $f\text{-}dist_{(\mathcal{E},\mathcal{A})}(\sigma)$ for $f\text{-}dist_{(\mathcal{E},\mathcal{A})}(\sigma, \delta_{\bar{q}(\mathcal{E}||\mathcal{A})})$.*

We can see next definition of f-implementation as the incapacity of an environment to distinguish two automata if it uses only information filtered by the perception function f.

▶ **Definition 29** (f-implementation). *Let $f_{(.,.)}$ be an insight-function. Let S be a scheduler schema. We say that $\mathcal{A}$ f*-implements $\mathcal{B}$ according to S*, noted $\mathcal{A} \leq_0^{S,f} \mathcal{B}$, if $\forall \mathcal{E} \in env(\mathcal{A}) \cap env(\mathcal{B})$, $\forall \sigma \in S(\mathcal{E}||\mathcal{A})$, $\exists \sigma' \in S(\mathcal{E}||\mathcal{B})$, $f\text{-}dist_{(\mathcal{E},\mathcal{A})}(\sigma) \equiv f\text{-}dist_{(\mathcal{E},\mathcal{B})}(\sigma')$, i.e. $\forall C \in supp(f\text{-}dist_{(\mathcal{E},\mathcal{A})}(\sigma)) \cup supp(f\text{-}dist_{(\mathcal{E},\mathcal{B})}(\sigma'))$, $f\text{-}dist_{(\mathcal{E},\mathcal{A})}(\sigma)(C) = f\text{-}dist_{(\mathcal{E},\mathcal{B})}(\sigma')(C)$.*

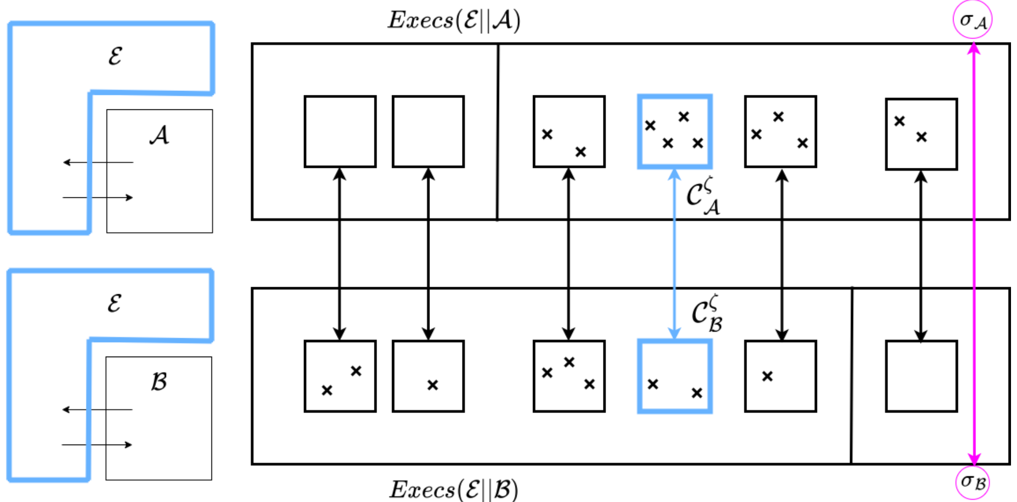


Figure 3 We say that $\mathcal{A}$ implements $\mathcal{B}$ if no environment $\mathcal{E}$ is able to distinguish $\mathcal{A}$ from $\mathcal{B}$, i.e. $\forall \sigma_{\mathcal{A}} \in schedulers(\mathcal{E}||\mathcal{A})$, $\exists \sigma_{\mathcal{B}} \in schedulers(\mathcal{E}||\mathcal{B})$ (linked by pink arrow) s.t. every pair of corresponding classes of equivalence of executions, related to the same perception by the environment (e.g. $(C^{\zeta}_{\mathcal{A}}, C^{\zeta}_{\mathcal{B}})$ in blue for perception ζ) are equiprobable, i.e. $f\text{-}dist_{(\mathcal{E},\mathcal{A})}(\sigma_{\mathcal{A}})(\zeta) = f\text{-}dist_{(\mathcal{E},\mathcal{B})}(\sigma_{\mathcal{B}})(\zeta)$.

We can restate classic theorem of (horizontal) substitutability of implementation in a quite general form.

▶ **Theorem 30** (Implementation substitutability). *Let $f_{(.,.)}$ be a perception-function. Let S be a scheduler schema. Let $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$, $\mathcal{B}$, $\mathcal{B}_1$, $\mathcal{B}_2$ be some PSIOA (resp. PCA).*

- *(Composability) If $\mathcal{A}_1 \leq_0^{S,f} \mathcal{A}_2$ and $\mathcal{B}$ is partially compatible with $\mathcal{A}_1$ and $\mathcal{A}_2$, then $\mathcal{B}||\mathcal{A}_1 \leq_0^{S,f} \mathcal{B}||\mathcal{A}_2$.*
- *(Transitivity) If $\mathcal{A}_1 \leq_0^{S,f} \mathcal{A}_2$ and $\mathcal{A}_2 \leq_0^{S,f} \mathcal{A}_3$, then $\mathcal{A}_1 \leq_0^{S,f} \mathcal{A}_3$.*
- *(Substitutability) If $\mathcal{A}_1 \leq_0^{S,f} \mathcal{A}_2$, $\mathcal{B}_1 \leq_0^{S,f} \mathcal{B}_2$, and both $\mathcal{B}_1$ and $\mathcal{B}_2$ are partially compatible with both $\mathcal{A}_1$ and $\mathcal{A}_2$, then $\mathcal{A}_1||\mathcal{B}_1 \leq_0^{S,f} \mathcal{A}_2||\mathcal{B}_2$.*

Substitutability constitutes one of the most important properties that an implementation relation should satisfy, since it allows to reason in a modular way and avoid overwhelming monolithic proof of correctness.

7 Dynamic vertical substitutability

In previous section, we have stated the classic horizontal substitutability of implementation relation, which allows us to replace an idealized abstract object by its concrete implementation without losing hyper-properties. In this section, we informally describe the main result of this paper: the dynamic vertical substitutability of p-implementation.

Informally, if (1) $X_{\mathcal{A}}$ and $X_{\mathcal{B}}$ are PCA that differ only on the fact that $\mathcal{B}$ supplants $\mathcal{A}$ in $X_{\mathcal{B}}$ and (2) $\mathcal{A}R\mathcal{B}$ for some preorder R implies (3) $X_{\mathcal{A}}RX_{\mathcal{B}}$, then we say that R is monotonic w.r.t. PSIOA creation/destruction. Monotonicity of implementation w.r.t. PSIOA creation/destruction is the main contribution of the paper.

▶ **Definition 31** ((Informal) corresponding w.r.t. $\mathcal{A}$, $\mathcal{B}$). *Intuitively, $X_{\mathcal{A}}$ and $X_{\mathcal{B}}$ are corresponding w.r.t. $\mathcal{A}$, $\mathcal{B}$ if they differ only in that $X_{\mathcal{A}}$ dynamically creates and destroys automaton $\mathcal{A}$ instead of creating and destroying automaton $\mathcal{B}$ as $X_{\mathcal{B}}$ does. Some technical minor assumptions have to be verified:*

- $X_{\mathcal{A}}$ *is* $\mathcal{A}$*-conservative and* $X_{\mathcal{B}}$ *is* $\mathcal{B}$*-conservative: Each state of* $X_{\mathcal{A}}$ *(resp.* $X_{\mathcal{B}}$*) is perfectly defined by its configuration deprived of sub-automaton* $\mathcal{A}$ *(resp.* $\mathcal{B}$*) and external actions of* $\mathcal{A}$ *(resp.* $\mathcal{B}$*) are not hidden.*
- $X_{\mathcal{A}}$ *is* $\mathcal{A}$*-creation explicit and* $X_{\mathcal{B}}$ *is* $\mathcal{B}$*-creation explicit: the creation of* $\mathcal{A}$ *and* $\mathcal{B}$ *respectively, are equivalent to the triggering of an action in a dedicated set.*
- $config(X_{\mathcal{A}})(\bar{q}_{X_{\mathcal{A}}}) \lhd_{AB} config(X_{\mathcal{B}})(\bar{q}_{X_{\mathcal{B}}})$*: The associated configuration of respective start states are identical except that the automaton* $\mathcal{B}$ *supplants* $\mathcal{A}$ *but with the same external signature.*
- $X_{\mathcal{A}}, X_{\mathcal{B}}$ *are creation&hiding-corresponding w.r.t.* $\mathcal{A}, \mathcal{B}$*: the two PCA hide some output actions and create some PSIOA in the same manner, excepting for the creation of* $\mathcal{B}$ *that supplants the creation of* $\mathcal{A}$*.*
- $\forall \mathcal{K} \in \{\mathcal{A}, \mathcal{B}\}$*,* $\forall q \in Q_{X_{\mathcal{K}}}$ *, for every* $\mathcal{K}$*-exclusive action a at state q,* $created(X_{\mathcal{K}})(q)(a) = \emptyset$*, where a* $\mathcal{K}$*-exclusive action is an action which is in the signature of sub-automaton* $\mathcal{K}$ *only.*

We would like to state the monotonicity of p-implementation, but it holds only for a certain class of schedulers, so-called *creation-oblivious* that does not take past internal behaviours of sub-automata into account to outputs the next action.

▶ **Definition 32** ((Informal) creation-oblivious scheduler)**.** *Let* $\tilde{A}$ *be a PSIOA,* $\tilde{W}$ *be a PCA,* $\tilde{\sigma} \in schedulers(\tilde{W})$*. We say that* $\tilde{\sigma}$ *is* $\mathcal{A}$-creation oblivious *if for every triplet* $(\tilde{\alpha}_1, \tilde{\alpha}_2, \tilde{\alpha}_3)$ *s.t. (1)* $lstate(\tilde{\alpha}_1) = lstate(\tilde{\alpha}_2) = fstate(\tilde{\alpha}_3)$ *and (2)* $\tilde{\alpha}_1$ *and* $\tilde{\alpha}_2$ *differ only on* $\mathcal{A}$*-exclusive actions and internal states of sub-automaton* $\mathcal{A}$*, then (3)* $\tilde{\sigma}(\tilde{\alpha}_1^\frown \tilde{\alpha}_3) = \tilde{\sigma}(\tilde{\alpha}_2^\frown \tilde{\alpha}_3)$*.*

Formal definitions of two last concepts are available in the extended version. It is crucial to limit the power of the scheduler to reduce the measure of a class of comportment as a function of measures of classes of shorter comportment where no creation of $\mathcal{A}$ or $\mathcal{B}$ occurs excepting potentially at very last action. This reduction is more or less necessary to obtain monotonicity of implementation relation:

▶ **Theorem 33** (p-implementation monotonicity)**.** *Let* $\mathcal{A}, \mathcal{B} \in Autids$*,* $X_{\mathcal{A}}$ *and* $X_{\mathcal{B}}$ *be PCA corresponding w.r.t.* $\mathcal{A}$*,* $\mathcal{B}$*. Let S the schema of creation-oblivious scheduler and* $p = proj_{(.,.)}$*. If* $\mathcal{A} \leq_0^{S,p} \mathcal{B}$*, then* $X_{\mathcal{A}} \leq_0^{S,p} X_{\mathcal{B}}$

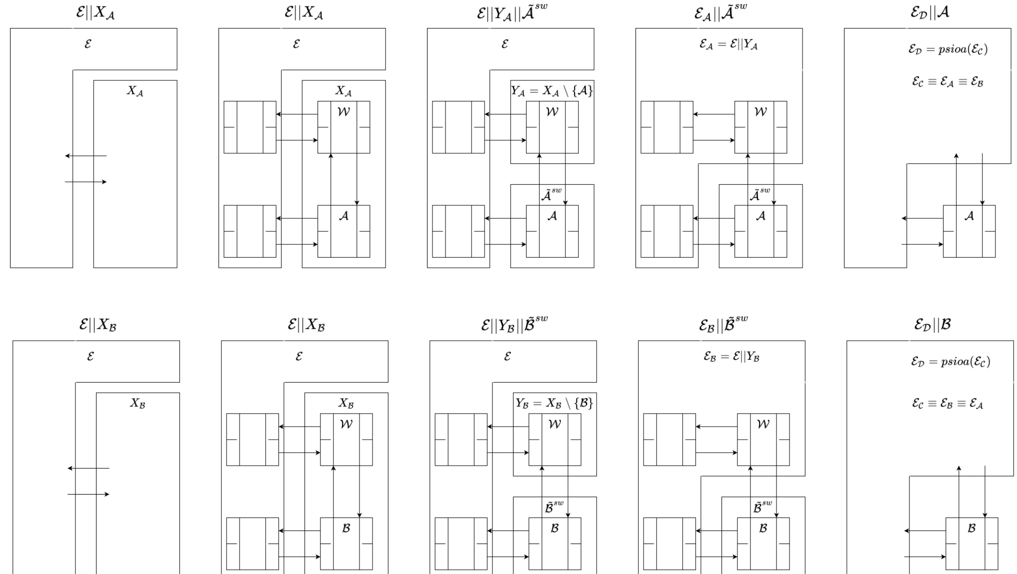
Proof Sketch. First, we defined the notion of executions-matching to capture the idea that two automata have the same "comportment" along some corresponding executions. Basically an executions-matching from a PSIOA $\mathcal{A}$ to a PSIOA $\mathcal{B}$ is a morphism $f^{ex} : Execs'_{\mathcal{A}} \to Execs(\mathcal{B})$ where $Execs'_{\mathcal{A}} \subseteq Execs(\mathcal{A})$. This morphism preserves some properties along the pair of matched executions: signature, transition, ... in such a way that for every pair $(\alpha, \alpha') \in Execs(\mathcal{A}) \times Execs(\mathcal{B})$ s.t. $\alpha' = f^{ex}(\alpha)$, $\epsilon_{\sigma}(\alpha) = \epsilon_{\sigma'}(\alpha')$ for every pair of schedulers (σ, σ') (so-called *alter ego*) that are "very similar" in the sense they take into account only the "structure" of the argument to return a sub-probability distribution, i.e. $\alpha' = f^{ex}(\alpha)$ implies $\sigma(\alpha) = \sigma'(\alpha')$. When the executions-matching is a bijection function from $Execs(\mathcal{A})$ to $Execs(\mathcal{B})$, we say $\mathcal{A}$ and $\mathcal{B}$ are semantically-equivalent (they differ only syntactically). Second, we defined the notion of a PCA $X_{\mathcal{A}}$ deprived of a PSIOA $\mathcal{A}$, noted $(X_{\mathcal{A}} \setminus \{\mathcal{A}\})$. Such an automaton corresponds to the intuition of a similar automaton where $\mathcal{A}$ is systematically removed from the configuration of the original PCA. Thereafter we shew that under technical minor assumptions $X_{\mathcal{A}} \setminus \{\mathcal{A}\}$ and $\tilde{\mathcal{A}}^{sw}$ are partially-compatible where $\tilde{\mathcal{A}}^{sw}$ and $\mathcal{A}$ are semantically equivalent. In fact $\tilde{\mathcal{A}}^{sw}$ is the simpleton wrapper of $\mathcal{A}$, that is a PCA

that only owns $\mathcal{A}$ in its attached configuration. Then we shew that there is an (incomplete) execution-matching from $X_{\mathcal{A}}$ to $(X_{\mathcal{A}} \setminus \{\mathcal{A}\})||\tilde{\mathcal{A}}^{sw}$. The domain of this executions-matching is the set of executions where $\mathcal{A}$ is not (re-)created before very last action. After this, we always try to reduce any reasoning on $X_{\mathcal{A}}$ (resp. $X_{\mathcal{B}}$) on a reasoning on $(X_{\mathcal{A}} \setminus \{\mathcal{A}\})||\tilde{\mathcal{A}}^{sw}$ (resp. $(X_{\mathcal{B}} \setminus \{\mathcal{B}\})||\tilde{\mathcal{B}}^{sw}$). We shew that, under certain reasonable technical assumptions (captured in the definition of corresponding PCA w.r.t. $\mathcal{A}$, $\mathcal{B}$), $(X_{\mathcal{A}} \setminus \{\mathcal{A}\})$ and $(X_{\mathcal{B}} \setminus \{\mathcal{B}\})$ are semantically-equivalent. We can note Y an arbitrary PCA semantically-equivalent to $(X_{\mathcal{A}} \setminus \{\mathcal{A}\})$ and $(X_{\mathcal{B}} \setminus \{\mathcal{B}\})$. Finally, a reasoning on $\mathcal{E}||X_{\mathcal{A}}$ (resp. $\mathcal{E}||X_{\mathcal{B}}$) can be reduced to a reasoning on $\mathcal{E}'||\tilde{\mathcal{A}}^{sw}$ (resp. $\mathcal{E}'||\tilde{\mathcal{B}}^{sw}$) with $\mathcal{E}' = \mathcal{E}||Y$. Since $\tilde{\mathcal{A}}^{sw}$ implements $\tilde{\mathcal{B}}^{sw}$, we have already some results on $\mathcal{E}'||\tilde{\mathcal{A}}^{sw}$ and $\mathcal{E}'||\tilde{\mathcal{B}}^{sw}$ and so on $\mathcal{E}||X_{\mathcal{A}}$ and $\mathcal{E}||X_{\mathcal{B}}$. However, this reduction, represented in figure 4, is valid only for the subset of executions without creation of neither $\mathcal{A}$ nor $\mathcal{B}$ before very last action. Ideally, we would like to decompose an "aggregated" class of perception, with arbitrary number of creations/destructions of $\mathcal{A}$ (resp. $\mathcal{B}$)), into "atomic" classes of perception without creation/destruction of $\mathcal{A}$ (resp. $\mathcal{B}$)) before last action. Some technical precautions have to be taken to be allowed to paste these fragments together to finally say that $\mathcal{A}$ implements $\mathcal{B}$ implies $X_{\mathcal{A}}$ implements $X_{\mathcal{B}}$. In fact, such a pasting is generally not possible for a fully information online scheduler. This observation motivated us to introduce the creation-oblivious scheduler, to manipulate independent atomic classes of perception. We proved monotonicity of external behaviour inclusion for schema of creation oblivious scheduler. Surprisingly, the fully-offline task-scheduler introduced in [3] (slightly modified to be adapted to dynamic setting) is not creation-oblivious and so does not allow monotonicity of implementation. ◀

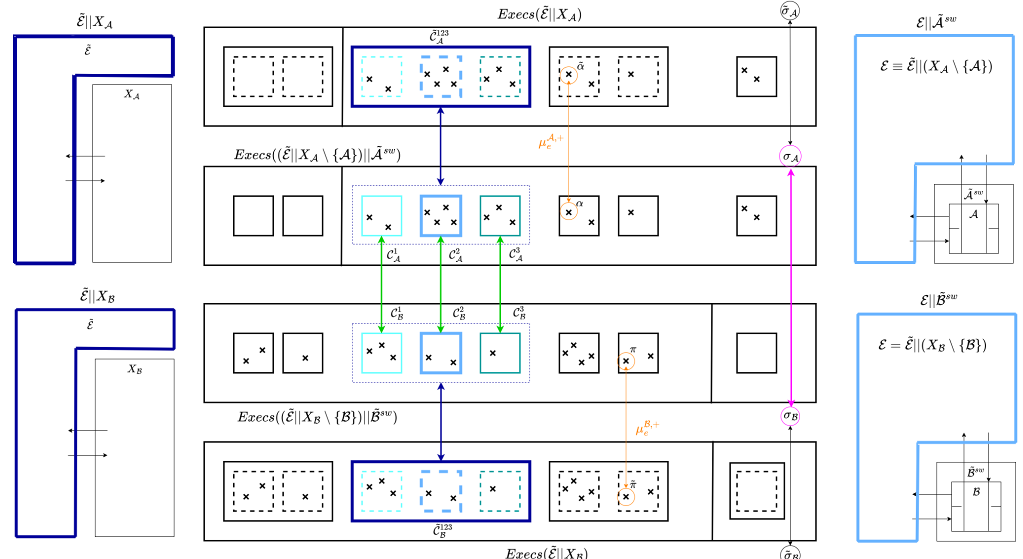
8 Conclusion

We have extended *dynamic I/O Automata* formalism of Attie & Lynch [2] to probabilistic setting in order to cope with emergent distributed systems such as peer-to-peer networks, robot networks, adhoc networks or blockchains. Our formalism includes operators for parallel composition, action hiding, action renaming, automaton creation and use a refined definition of probabilistic configuration automata in order to cope with dynamic actions. The key result of our framework is as follows: the implementation of probabilistic configuration automata is monotonic to automata creation and destruction. That is, if systems $X_{\mathcal{A}}$ and $X_{\mathcal{B}}$ differ only in that $X_{\mathcal{A}}$ dynamically creates and destroys automaton $\mathcal{A}$ instead of creating and destroying automaton $\mathcal{B}$ as $X_{\mathcal{B}}$ does, and if $\mathcal{A}$ implements $\mathcal{B}$ (in the sense they cannot be distinguished by any external observer), then $X_{\mathcal{A}}$ implements $X_{\mathcal{B}}$. This results is particularly interesting in the design and refinement of components and subsystems in isolation. In our construction we exhibit the need of considering only *creation-oblivious* schedulers in the implementation relation, i.e. a scheduler that, upon the (dynamic) creation of a sub-automaton $\mathcal{A}$, does not take into account the previous internal behaviour of $\mathcal{A}$ to output (randomly) a transition.

As future work we plan to extend the composable secure-emulation of Canetti et al. [5] to dynamic settings. This extension is necessary for formal verification of protocols combining probabilistic distributed systems and cryptography in dynamic settings (e.g. blockchains, secure distributed computation, cybersecure distributed protocols etc).



(a) The figure represents successive steps to reduce the problem of an environment $\mathcal{E}$ that tries to distinguish two PCA $X_{\mathcal{A}}$ and $X_{\mathcal{B}}$ (represented at first column) to a problem of an environment $\mathcal{E}_{\mathcal{D}}$ that tries to distinguish the automata $\mathcal{A}$ and $\mathcal{B}$ (represented at last column).



(b) The figure represents the homomorphism enabling the reduction reasoning, for set of executions that do not create neither $\mathcal{A}$ nor $\mathcal{B}$ before last action. For every environment $\mathcal{E}$, For every scheduler $\sigma_{\mathcal{A}}$, there exists a corresponding scheduler $\sigma_{\mathcal{B}}$ (mapped with pink arrow) s.t. for every possible perception ζ (represented in light blue), the probability to observe ζ is the same for $\mathcal{E}$ in each world. There is an homomorphism $\mu_e^{\mathcal{A},+}$ (orange arrow) between $\tilde{\mathcal{E}}||X_{\mathcal{A}}$ and $\mathcal{E}||\tilde{\mathcal{A}}^{sw}$ (and similarly for $X_{\mathcal{B}}$ and $\tilde{\mathcal{B}}^{sw}$) s.t. for every scheduler $\tilde{\sigma}_{\mathcal{A}}$, alter-ego of $\sigma_{\mathcal{A}}$, the measure of each corresponding perception is preserved. Hence, for every environment $\tilde{\mathcal{E}}$, for every scheduler $\tilde{\sigma}_{\mathcal{A}}$, there exists a corresponding scheduler $\tilde{\sigma}_{\mathcal{B}}$ s.t. for every possible perception $\tilde{\zeta}$ (represented in dark blue), the probability to observe $\tilde{\zeta}$ is the same for $\tilde{\mathcal{E}}$ in each world.

Figure 4 homomorphism-based-proof.

References

1. Edward A. Ashcroft. Proving assertions about parallel programs. *J. Comput. Syst. Sci.*, 10(1):110–135, 1975. doi:10.1016/S0022-0000(75)80018-3.
2. Paul C. Attie and Nancy A. Lynch. Dynamic input/output automata: A formal and compositional model for dynamic systems. *Inf. Comput.*, 249:28–75, 2016. doi:10.1016/j.ic.2016.03.008.
3. Ran Canetti, Ling Cheung, Dilsun Kaynar, Moses Liskov, Nancy Lynch, Olivier Pereira, and Roberto Segala. Task-Structured Probabilistic {I/O} Automata. *Journal of Computer and System Sciences*, 94:63—-97, 2018. doi:10.1016/j.jcss.2017.09.007.
4. Ran Canetti, Ling Cheung, Dilsun Kirli Kaynar, Moses D. Liskov, Nancy A. Lynch, Olivier Pereira, and Roberto Segala. Using probabilistic I/O automata to analyze an oblivious transfer protocol. *IACR Cryptol. ePrint Arch.*, page 452, 2005. URL: http://eprint.iacr.org/2005/452.
5. Ran Canetti, Ling Cheung, Dilsun Kirli Kaynar, Nancy A. Lynch, and Olivier Pereira. Compositional security for task-pioas. In *20th IEEE Computer Security Foundations Symposium, CSF 2007, 6-8 July 2007, Venice, Italy*, pages 125–139. IEEE Computer Society, 2007. doi:10.1109/CSF.2007.15.
6. Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theor. Comput. Sci.*, 777:155–183, 2019. doi:10.1016/j.tcs.2019.02.001.
7. Pierre Civit and Maria Potop-Butucaru. Probabilistic dynamic input output automata (extended version). Cryptology ePrint Archive, Paper 2021/798, 2021. doi:10.4230/LIPIcs.DISC.2022.20.
8. Pierre Civit and Maria Potop-Butucaru. Brief announcement: Composable dynamic secure emulation. In Kunal Agrawal and I-Ting Angelina Lee, editors, *SPAA '22: 34th ACM Symposium on Parallelism in Algorithms and Architectures, Philadelphia, PA, USA, July 11 - 14, 2022*, pages 103–105. ACM, 2022. doi:10.1145/3490148.3538562.
9. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
10. Richard M. Karp and Raymond E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, 1969. doi:10.1016/S0022-0000(69)80011-5.
11. Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977. doi:10.1109/TSE.1977.229904.
12. Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. A theory of atomic transactions. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 326 LNCS:41–71, 1988. doi:10.1007/3-540-50171-1_3.
13. Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. doi:10.1007/3-540-10235-3.
14. Rocco De Nicola and Roberto Segala. A process algebraic view of input/output automata. *Theor. Comput. Sci.*, 138(2):391–423, 1995. doi:10.1016/0304-3975(95)92307-J.
15. Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976. doi:10.1007/BF00268134.
16. C.A. Petri. *Kommunikation mit Automaten.* PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962.
17. Martin L. Puterman. *Markov decision processes: discrete stochastic dynamic programming.* Wiley series in probability and mathematical statistics. John Wiley & Sons, 1 edition, 1994.
18. Alejandro Ranchal-Pedrosa and Vincent Gramoli. Platypus: Offchain protocol without synchrony. In Aris Gkoulalas-Divanis, Mirco Marchetti, and Dimiter R. Avresky, editors, *18th IEEE International Symposium on Network Computing and Applications, NCA 2019, Cambridge, MA, USA, September 26-28, 2019*, pages 1–8. IEEE, 2019. doi:10.1109/NCA.2019.8935037.
19. Roberto Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, Massachusettes Institute of technology, 1995.

20 Frits W. Vaandrager. On the relationship between process algebra and input/output automata. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, pages 387–398. IEEE Computer Society, 1991. doi:10.1109/LICS.1991.151662.

21 Kazuki Yoneyama. Formal modeling of random oracle programmability and verification of signature unforgeability using task-pioas. *Int. J. Inf. Sec.*, 17(1):43–66, 2018. doi:10.1007/s10207-016-0352-y.

How to Wake up Your Neighbors: Safe and Nearly Optimal Generic Energy Conservation in Radio Networks

Varsha Dani ✉
Department of Computer Science, Rochester Institute of Technology, NY, USA

Thomas P. Hayes ✉
Department of Computer Science, University at Buffalo, NY, USA

Abstract

Recent work [7, 8, 11] has shown that it is sometimes feasible to significantly reduce the energy usage of some radio-network algorithms by adaptively powering down the radio receiver when it is not needed. Although past work has focused on modifying specific network algorithms in this way, we now ask the question of whether this problem can be solved in a generic way, treating the algorithm as a kind of black box.

We are able to answer this question in the affirmative, presenting a new general way to modify arbitrary radio-network algorithms in an attempt to save energy. At the expense of a small increase in the time complexity, we can provably reduce the energy usage to an extent that is provably nearly optimal within a certain class of general-purpose algorithms.

As an application, we show that our algorithm reduces the energy cost of breadth-first search in radio networks from the previous best bound of $2^{O(\sqrt{\log n})}$ to polylog(n), where n is the number of nodes in the network

A key ingredient in our algorithm is hierarchical clustering based on additive Voronoi decomposition done at multiple scales. Similar clustering algorithms have been used in other recent work on energy-aware computation in radio networks, but we believe the specific approach presented here may be of independent interest.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Radio Networks, Low Energy Computation, Clustering

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.16

Related Version *Full Version*: https://arxiv.org/abs/2205.12830

1 Introduction

For large networks of tiny sensors equipped with radio transceivers, the energy used in communication can be a bottleneck cost. Although it has become standard practice to measure the cost of communication in terms of the number of messages or bits sent, it has been pointed out (see, for instance, [2, 30]) that the cost of using the receiver to listen for messages is often comparable to, or even more than, the cost of transmission. This becomes even more true as the size of the devices is scaled down.

Obviously, this fact suggests that we should try hard to avoid using our receiver to listen for messages except at times when one is being sent. However, this is easier said than done! In general, effective economization of receiver use probably requires redesigning our communication protocols from the ground up. As an example, previous work by Dani, Gupta, Hayes and Pettie [11] on solving the maximal matchings problem on a radio network with low energy expenditure, presented an algorithm that was very specific to its problem, in the sense that the algorithm is able to cleverly combine efficient message delivery with energy conservation and residual degree manipulation, all at once.

© Varsha Dani and Thomas P. Hayes;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 16; pp. 16:1–16:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Nevertheless, it is tempting to ask whether, in spite of this, some general-purpose technique can be used to minimize receiver usage when no messages are being sent nearby. In this paper we give a positive answer to this question. More precisely, we present a generic way to convert any given protocol into one that attempts to use its receiver more wisely, in a way tailored to the activity pattern of the given algorithm. The amount of energy saved depends on properties of the algorithm being simulated. For some protocols, our construction will actually make them less energy efficient. However, for at least a few protocols of interest, our technique improves the overall energy cost on size n networks from $\text{poly}(n)$ to $\text{polylog}(n)$. Moreover, there is a sense in which our algorithm is nearly the best possible, as we shall see.

This potential decrease in overall energy cost is not completely free. Our algorithm incurs modest (up to a $\text{polylog}(n)$ factor) penalties in terms of overall running time, (sent) message complexity, and local computation. These costs should be weighed against the possible benefits before deciding whether to deploy our algorithm in a particular context.

We use a simple abstract model of distributed computation on Radio Networks [10], specifically the variant introduced in [7]. In this model, local computation is treated as free, and in each of a series of globally synchronized timesteps, each processor decides whether to SEND, LISTEN, or SLEEP. Sending and Listening cost 1 unit of energy, but sleeping is free. Messages can be up to $O(\log n)$ bits long. In this setting, we try to simultaneously minimize the total time complexity of our protocol, as well as the maximum per-node energy usage.

The motivation for our new construction comes from the structure of the simplest possible network algorithm: a naive broadcast protocol, in which a message is repeatedly transmitted across the network at a rate of unit distance per timestep. Considering the behavior of the "active set," that is, the nodes that are actually transmitting or receiving a message in a given timestep, we see that it behaves like a ripple moving across a pool of water, or the frontier of a parallel BFS algorithm, or even a light-cone propagating across spacetime.

One aspect of this is that the active set moves with a known finite speed, which can potentially allow us to shut down nodes when the active set is known to be far away from them. In particular, any time a node is at distance at least d from the active set, it can afford to sleep for at least d timesteps, because information cannot be transmitted faster than one edge per timestep. Of course, to use this in practice, our hypothetical node needs some way of getting advance warning of how far it is from the active set.

We achieve this by organizing nodes into clusters that spread information about the location of the active set faster than the "speed of light" in our simulated protocol. Or, to put it more prosaically, at the cost of increased latency, our network is able to transmit advance notice of the active set before it arrives. To save as much energy as possible, we try to do this at all distance scales simultaneously. Thus, while nodes whose distance to the active set is only a few dozen may only know they can safely sleep for a dozen timesteps; at the same time, nodes whose distance to the active set is in the thousands may know they can safely sleep for hundreds of timesteps. At a high level, this sounds very similar to the approach used by Chang, Dani, Hayes and Pettie [8] for reducing the energy cost of BFS in the same model. Indeed, the main difference in our current approach is in the way the clusters at different scales are generated and communicate with each other. As a result of these differences, we are able to exponentially improve on the energy usage of BFS, while at the same time, being immediately applicable to a broader range of algorithms.

We present a new general-purpose algorithm which we call SAF Simulation (see Algorithm 2) after its mnemonic, "Sleep when Activity is Far." SAF Simulation accomplishes both the goal of building layered cluster decompositions, as well as using this structure to effectively simulate a large class of algorithms. Our main result can be summarized (at a high level) as

▶ **Theorem 1.** *For every Radio Network algorithm, A, SAF(A) runs in time* $O(\text{polylog}(n)\,\text{TIME}(A))$*, and with probability* $1 - 1/\text{poly}(n)$ *produces the same output as* A*. Moreover, its energy cost satisfies, for every vertex* v*,*

$$\text{ENERGY}(\text{SAF}(A), v) \leq \text{polylog}(n)\text{OPT}(A, v),$$

where OPT *is the least possible energy cost for a safe, synchronized, one-pass generic simulation algorithm.*

A more precise (and technical) statement appears in Theorem 14. Some specific problems that can be solved by Algorithm 2 for only polylog(n) per-node energy include: Broadcast, BFS, Leader Election, and Approximate Counting. Loosely speaking, the same can be said for any algorithm that does the bulk of its computation in polylog(n) BFS-like sweeps across the graph. In all cases, this represents an exponential improvement in energy usage, as compared to the naive implementation.

Comparison to Previous Work

Our main tool for clustering our nodes is the ESTCluster algorithm of Miller, Peng and Xu [26] (see also [27].) This results in clusters that are the cells of a kind of generalized Voronoi diagram, as we discuss in Section 3.3. These clusters were used previously in the context of low energy computation in the same radio network model by Chang, Dani, Hayes and Pettie [8] for the problem of constructing breadth-first search.

In that work, they estimated long-range distances by recursively solving BFS on their level-1 cluster graph. This has the apparent advantage that, in the resulting hierarchy of clusters, each cluster is a coarsening of the previous lower-level cluster. However, it has the disadvantage that the distortion of distances accumulates multiplicatively as one moves up levels. This limits its usefulness for putting processors to Sleep, since one cannot risk missing the arrival of the active set.

In the current work, by contrast, instead of recursively constructing clusters of clusters, we instead run the ESTCluster algorithm on the base graph with different radius parameters, to generate our entire hierarchy of clusters. An apparent disadvantage of this is that the resulting clusterings are not coarsenings and refinements of each other. Fortunately, it turns out that we do not need this property. Although it would be possible to use a sort of rounding to replace our clusterings with ones that are nested, there are algorithmic disadvantages to doing so. Specifically, one nice thing about the generalized Voronoi cells created by the ESTCluster algorithm is that they are star-shaped, so the entire cluster is connected by a BFS spanning tree rooted at the cluster center. This property would be lost if redefined the clustering to be a coarsening of the lower-level clusterings. This would appear to necessitate neighboring clusters to assist in what was previously intra-cluster communications.

Other related work on energy complexity in radio networks

A number of authors have studied energy complexity on *single-hop* radio networks. Here the energy model is still that listening costs as much as sending, while sleeping and local computation are free, but the underlying graph is a clique. When nodes choose to transmit they are broadcasting their message to the entire graph, or at least anyone who is listening, and the main issue is just how to resolve contention for the channel. Moreover, in much of this work, there is no bound on message sizes. In this model, Nakano and Olariu [28] gave an $O(\log \log n)$ energy algorithm for selecting distinct IDs in $\{1, \ldots, |V| = n\}$. Bender, Kopelowitz, Pettie

and Young [3] showed that $O(\log(\log^* n))$ energy suffices for all devices to send messages if there is collision-detection available on the channel. Chang, Kopelowitz, Pettie, Wang and Zhan [5] showed this to be optimal, and studied the problems of Leader Election and Approximate Counting, both with and without randomization, with collision detection. They also studied tradeoffs between time, energy and error probability. Similar tradeoffs were also studied by Kardas et al. [21]. Jurdzinski, Kutylowski and Zatopianski [16, 15, 17, 18, 19]. studied the Leader Election and Approximate Counting problems in the absence of collision detection.

The single-hop notion of energy complexity was extended to arbitrary networks by Chang, Dani, Hayes, He, Li, and Pettie [7] where they studied the Broadcast problem, with and without collision detection, and with and without randomization. The energy complexity was shown to be polylog(n) in all cases. However, their broadcast algorithm did not transmit messages along shortest paths. Later Chang, Dani, Hayes and Pettie [8] gave a $2^{O(\sqrt{\log n})}$ algorithm for BFS. They also showed that the diameter of a network is hard to approximate to within factor 2 using sublinear energy.

Other models of energy cost have also been studied in the radio network literature. Gasnieniec et al. [12], Berenbrink et al. [4] and Klonowski and Pajak [23], studied broadcast and gossiping problems under a cost model where the goal is to minimize the worst case number of transmissions per device. Klonowski and Sulkowska [24] defined a distributed model in which devices can transmit messages at varying power levels, which can be chosen online. The devices in question are at random locations in the d-dimensional cube. A number of works [25, 20, 13, 22] also considered robustness of low-energy algorithms against a jamming attack. Here an adversary attempts to foil the devices' attempts to communicate by making noise on the channel. The goal here is not precisely low-energy use, but rather "resource competitive" energy use, where the processors combined cost for getting messages through should be commensurate with the adversary's budget for wreaking havoc.

A related model in distributed computing on wired networks is the Sleeping Model of Chatterjee *et al.* [9]. In their model, which is a variant of CONGEST, nodes can also save energy by going to sleep, in which case they may miss incoming messages. However, their model is considerably more powerful than its radio network analog, in that they allow a node to send or receive distinct messages from each of its awake neighbors in a single round.

Organization of the Paper

Section 2 contains a more detailed introduction to the radio network model. Section 3 discusses our method of cluster formation, and related issues. In Section 4, we discuss the problem of simulating radio network algorithms, and give a formal definition of what we mean by "optimal" simulation. In Section 5, we describe our simulation algorithm, and give formal statements of our main results. In Section 6, we apply our simulation algorithm to the breadth-first search problem, showing it can be solved in polylog energy. Due to space considerations, all proofs have been relegated to the Appendix. A longer version of this paper is available at arXiv:2205.12830[cs.DC].

2 Preliminaries: the Low Energy Radio Network Model

The Network

We assume there is a communication network on an arbitrary, undirected connected graph $G = (V, E)$. At each node in G, there is a processor equipped with a transmitter and receiver to communicate with other nodes. There is an edge between nodes u and v in the graph if u and v are within transmission range of each other.

The processors are identical, except for having unique IDs. They do not know the underlying graph G; indeed we assume that initially they do not even know their neighbors in the graph. A processor will become aware of (and remember) any neighbor once it has communicated with it. The processors do have a shared estimate on the size of the graph; that is, a number $n \geq |V|$ is known to all the processors and may be used in any algorithms they run. Accuracy of this estimate is not required for correctness of the algorithm, but the time and energy usage of algorithms will depend on it. Additionally we assume that the processors can generate independent streams of random bits. There is no shared randomness. **For randomized algorithms, we assume that each node does all of its coin flipping prior to the beginning of the algorithm, generating a string of random bits to be read off and used at the appropriate time in the algorithm.** This is a standard construct for viewing a randomized algorithm as a deterministic algorithm with an additional random input.

Time

Time is divided into discrete, synchronous timesteps, and the processors agree on a time $t = 0$. In each timestep a processor can choose to do one of three actions: transmit, listen, or sleep. When a processor decides to transmit at time t, it sends a message of size $O(\log n)$ bits, whereupon it is potentially heard by any of its neighbors who happen to be listening at time t. We say "potentially" because there are a number of different models for what happens if a listening node encounters a collision, *i.e.* two or more of its neighbors are broadcasting messages in the timestep that it is listening. Nevertheless, at a minimum, we can say that a message travels from a node u to a neighbor v of u at time t if

- u decides to transmit at time t,
- v decides to listen at time t and
- no other neighbor of v decides to transmit at time t.

Collisions and Message Delivery

There are several different models for how to handle collisions, that is, what information does a listening node receive when two or more of its neighbors are sending messages in a single round? In the most permissive of these, the **Broadcast CONGEST model**, v receives all the messages sent by its neighbors, as if they were being sent on separate channels.

A more restrictive model is the **Collision Detection model (CD)** where, when a listener is next to more than one sender, a special "noise" message is received, which is distinguishable from silence (no neighbors sending), but carries no further information.

Another model of interest is the **No Collision Detection model (no-CD)**, which is even more restrictive: here, collisions are indistinguishable from silence.

Exponential backoff, introduced in the context of radio networks by Bar-Yehuda, Goldreich and Itai [1], is a commonly used technique for handling collisions. Among other uses, it can be used to eliminate the problem of collisions; see, for example, [6, 7, 8]. The basic idea is to simulate each timestep using $O(\log^2 n)$ timesteps, divided into $O(\log n)$ rounds of $O(\log n)$ timesteps each. A listener listens for the entire interval of length $O(\log^2 n)$ or until a message is received. In each round, senders flip coins at each timestep to decide whether to (re-)transmit, or to retire for the rest of the round, resulting in a constant probability that, in a particular round, there is at least one timestep in which a unique neighbor of the listener is sending, and therefore a message is successfully received, regardless of the collision model. The net result of this $O(\log^2 n)$-time backoff procedure is that, with probability $1 - 1/\text{poly}(n)$, every listener successfully receives a message, assuming at least one neighbor is a sender.

The fact that collisions can be handled in this manner inspires the definition of the following alternative message delivery model without collisions. In the "OR" model of message delivery, every time a node listens, it receives an arbitrary message sent by one of its neighbors in that timestep. The only exception is when no neighbor of the listening node chose to send in that timestep, in which case no message is received. For convenience, we will assume for the rest of this paper that we are working in the OR model of message delivery.

Energy Usage

We measure the cost of our algorithms in terms of their energy usage. We assume that a node incurs a cost of 1 energy unit each time that it decides to send or listen. When the node is sleeping there is no energy cost. We also assume that local computation is free. The goal of energy-aware computation is to design algorithms where the nodes can schedule sleep and communication times so that the per-node energy expenditure is small, ideally $\text{polylog}(n)$, without compromising the time complexity too much, i.e., the running time remains polynomial in n. In fact, our simulation algorithm does better: its time complexity is only a $\text{polylog}(n)$ factor greater than that of the simulated algorithm.

3 Cluster Graphs and Distance Approximation

In this section we introduce our framework for getting distance estimates that will be used by the nodes in the simulation algorithm.

3.1 Graph theoretic preliminaries

Let $G = (V, E)$ be a graph. Let $d : G \times G \to \mathbb{N}$ be the shortest path metric on G. Consider a partition $\mathcal{P} = (V_1, V_2, \ldots, V_k)$ of V into pairwise disjoint sets of vertices. We will call each V_i a "cluster". The *cluster graph*, also called the *quotient graph* and denoted $G/\mathcal{P}$, is a graph whose vertex set is the set of clusters, $\{V_1, \ldots V_k\}$. There is an edge in the cluster graph between V_i and V_j if there are vertices $u \in V_i$ and $v \in V_j$ such that $(u, v) \in E$.

Although the definition of a quotient graph does not require either G or the subgraphs induced by the clusters V_i to be connected, in our work we will assume that both are the case. It is easy to see that since G is connected, so is $G/\mathcal{P}$.

Given a partition $\mathcal{P}$, we will denote the cluster containing vertex u by $[u]$. Since the cluster graph is a graph, we can also define the shortest path distance metric on the cluster graph. We are interested in partitions where the individual clusters have comparable diameters, and the distances between clusters in the cluster graph are (approximately) scaled versions of the distances in the underlying graph.

3.2 Approximately Distance-Preserving Partitions

▶ **Definition 2.** *Let $G = (V, E)$ be a graph, and let $R, \alpha, \beta \geq 1$. Let $\mathcal{P}$ be a partition of V. Let d denote the distance metric in G and d^* denote the distance in $G/\mathcal{P}$. We say $\mathcal{P}$ is an (R, α, β)-approximately distance-preserving partition if*

- *for every pair of nodes u and v with $d(u, v) \leq R$, we have $d^*([u], [v]) \leq \alpha$*
- *for every pair of nodes u and v with $d^*([u], [v]) = 0$, (that is, u and v are in the same cluster) we have $d(u, v) \leq \beta R$.*

Taken together, the two halves of the above definition ensure that up to a multiplicative factor in the range $[1/\alpha, \beta]$, as well as possible rounding issues, we have, for all $u, v \in V$, that $d^*([u],[v]) \approx \frac{d(u,v)}{R}$. Specifically,

▶ **Lemma 3.** *Let $G = (V, E)$ be a graph, and $\mathcal{P}$ be an (R, α, β)-approximately distance-preserving partition of V. Let d and d^* denote the shortest path distance metrics in G and $G/\mathcal{P}$ respectively. Then, for all $u, v \in V$,*

$$\left\lfloor \frac{d(u,v)}{\beta R + 1} \right\rfloor \leq d^*([u],[v]) \leq \alpha \left\lceil \frac{d(u,v)}{R} \right\rceil.$$

A nice example of an approximately distance-preserving partition is for the square grid. Let G be the $n \times n$ square grid graph, and suppose R is a divisor of n. We partition V by rounding each point (x, y) down to the nearest multiple of R, $\left(R\lfloor \frac{x}{R} \rfloor, R\lfloor \frac{y}{R} \rfloor\right)$, and placing two vertices in the same cluster if they round to the same multiple of R. It is easy to see that this example is an $(R, 2, 2)$-approximately distance-preserving partition. Quite similar constructions can be done for all real $R \geq 1$, and for many other "homogeneous" graphs, such as lattice graphs of fixed dimension.

Although it may not be immediately obvious, such approximately distance-preserving partitions exist for all graphs, for all $R \geq 1$, and for $\alpha, \beta = O(\log n)$, as we shall see next.

3.3 Additive Weights Voronoi Diagrams and the MPX Algorithm

Additively Weighted Voronoi Decomposition (AWVD) (see, e.g., Phillips [29]), also known as hyperbolic Dirichlet tesselation, is a well-studied concept for real Euclidean domains. Start with a finite set of points in the plane, called generators, each of which is assigned a real-valued weight. Each point x in the plane is assigned to the generator g minimizing the sum $\|x - g\| - W(g)$, where $W(g)$ is the weight of g. After discarding any empty cells, this defines a partition of the plane into a finite collection of cells. When the weights are all zero, this corresponds to the usual notion of Voronoi diagram, and the boundaries of the cells are line segments and rays. For general weights, the boundaries of the cells are hyperbolic arcs. The cells are star-shaped with respect to their generators, but are generally not all convex.

For a finite graph, $G = (V, E)$, the analogous concept is a partition of V based on assigning a real-valued weight $W(v)$ to each vertex $v \in V$. We say that vertex u belongs to the cell generated by vertex v if $v = \arg\min_{v' \in V} d(v', w) - W(v')$. For convenience, we will assume that no two weights $W(v), W(v')$ differ by an integer, so that the cells are defined unambiguously. Two vertices belonging to the same cell is an equivalence relation, so each AWVD gives rise to a corresponding cluster graph.

Miller, Peng, and Xu [26] proposed a simple randomized graph-partitioning algorithm to obtain a decomposition with certain properties, specifically that the clusters have small diameter and only a small fraction of the edges of the graph are cut. In their construction, starting with a (common) parameter R, each vertex v independently samples a random variable $\delta_v \sim \text{Exponential}(1/R)$ from the exponential distribution with mean R. A cluster starts forming at each vertex v at time $-\delta_v$, and spreading through the graph at a uniform rate of one edge per time unit. Each vertex u either joins the first cluster to reach it before time $-\delta_u$ or starts its own cluster at time $-\delta_u$ if no other cluster has recruited it before that time. Haeupler and Wajc [14] showed that with minor modifications, this algorithm can be efficiently implemented in the Radio Network model.

We note that the MPX decomposition is, in fact, an AWVD where the weights $W(v)$ are independent exponentially-distributed random variables with mean R. We will now see that this decomposition has some good distance approximating properties.

As shown in [26], the clusters have diameter $O(R\log n)$. We state this more precisely:

▶ **Lemma 4.** *With probability at least* $1-\dfrac{1}{n^2}$*, the clusters in the MPX decomposition with parameter* R *have diameter at most* $3R\log n$*.*

Furthermore, Chang *et. al* [8] showed that not too many clusters are close to a single vertex. Specifically, if $\mathcal{P}$ is the partition determined by the MPX algorithm with parameter R, and $G^* = G/\mathcal{P}$ is the corresponding cluster graph, then Lemma 2.1 from [8] (translated into our notation) can be stated as follows:

▶ **Lemma 5.** *For every positive integer* j *and* $\ell > 0$*, the probability that the number of* G^**-clusters intersecting* $\mathrm{Ball}_G(v,\ell)$ *is more than* j *is at most* $(1-\exp(-2\ell/R))^j$*.*

▶ **Corollary 6.** *For any* $v \in V$*,* $\mathbf{P}\left(\textit{At most } 20\log n \textit{ clusters intersect } \mathrm{Ball}_G(v,\ell)\right) \geq 1-\frac{1}{n^2}$*.*

Combining Lemma 4 and Corollary 6, we have shown that

▶ **Proposition 7.** *With probability at least* $1-\frac{2}{n^2}$*, the partition* $\mathcal{P}$ *produced by the MPX algorithm with parameter* R *is a* $(R, 20\log n, 3\log n)$*-approximately distance-preserving partition*

3.4 Multi-Scale Clustering

▶ **Definition 8.** *Suppose, for* $1 \leq i \leq \ell$*, we have an approximately distance-preserving partition* $\mathcal{P}_i$ *with parameters* (R_i, α_i, β_i)*, where* $R_1 < \cdots < R_\ell$*. By convention, we extend this definition to the case* $i = 0$ *by setting* $R_0 = \alpha_0 = \beta_0 = 1$*, and letting the* $i = 0$ *partition be the partition of* V *into singleton vertex sets. Suppose further that for* $1 \leq i \leq \ell$*, we have*

$$\frac{R_{j+1}}{R_j} \geq (2\alpha_j + 1)(\beta_j + 1) - 1. \tag{1}$$

We call this a multi-scale clustering with parameters $(\mathbf{R}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = ((R_i), (\alpha_i), (\beta_i))_{i\in\{0,\dots\ell\}}$*.*

Notice that there is no requirement that the lower-level clusters be refinements of the higher-level clusters. Despite this, as we will see, multi-scale clusterings have a useful, albeit weaker, nesting property that controls the relationship between clusters at consecutive scales.

Suppose $\mathcal{P}_1, \mathcal{P}_2, \dots \mathcal{P}_\ell$ is a multi-scale clustering on G, with parameters $(\mathbf{R}, \boldsymbol{\alpha}, \boldsymbol{\beta})$. Let $G_1, \dots G_\ell$ be the corresponding cluster graphs, and $d_1, \dots, d_\ell$ the corresponding distance metrics. For $v \in V$, let $[v]_j$ denote the cluster containing v in $\mathcal{P}_j$. As usual, $\mathrm{Ball}_{G_j}([v]_j, r)$ denotes the ball of radius r in G_j. We will also need a notation for the vertices in the underlying graph G, whose clusters belong to this ball. To this end we define

$$B_j(v,r) = \{w \in V \mid d_j([w]_j, [v]_j) \leq r\}.$$

The following nesting property is true for all vertices, at all scales.

▶ **Lemma 9.** *Let* $1 \leq j < \ell$*,* $u \in V$*, and* $v \in \mathrm{Ball}_G(u, R_{j+1} - R_j)$*. That is,* $d(u,v) \leq R_{j+1} - R_j$*. Then* $B_j(v, 2\alpha_j) \subset B_{j+1}(u, 2\alpha_{j+1})$*.*

Although the only restriction imposed so far on the parameters $(\mathbf{R}, \boldsymbol{\alpha}, \boldsymbol{\beta})$ is that the R_js grow at a certain rate relative to the α_js and β_js, an interesting and natural special case is when (R_j) is an increasing geometric sequence, while sequences (α_j) and (β_j) are approximately constant.

▶ **Definition 10.** *In the special case when, for $1 \leq j \leq \ell$, $R_j = R^j$, for some $R > 12$, and $\alpha_j = \beta_j = \lfloor \sqrt{R}/2 \rfloor$, we call this a geometric multi-scale clustering with parameter R.*

We saw earlier that the MPX clustering with parameter R is, *w.h.p*, a $(R, 20\log n, 3\log n)$-approximate distance-preserving partition. Naturally, it is also a $(R, 20\log n, 12\log n)$ approximate distance-preserving partition. Setting $\alpha = \beta = 20\log n$ and $R = \Theta(\log^2 n)$ we can get a geometric multi-scale clustering by constructing MPX partitions for all scales $R, R^2, R^3 \ldots$

3.5 Simulating cluster-graph algorithms on the underlying graph

Chang et al. [7] defined protocols called UPCAST, DOWNCAST, and INTERCAST for communicating within and between clusters of nodes in a Radio Network. In the corresponding section of the Appendix, we describe these protocols at a high level, and remind the reader of some of their relevant properties.

4 Simulating Radio Network Algorithms

Let A be a radio network algorithm. When talking about the time and message complexity of A, since one does not charge for listening, one usually only specifies which timesteps A requires a node to SEND, with the implicit assumption that whenever a node is not SENDing it is LISTENing. Given any algorithm in this form, it is trivial to naively convert it to what we view here as the standard form, by making all LISTENs explicit. After such a naive conversion, the algorithm will have per-node energy cost equal to its time complexity, which is not good, unless the time complexity is already very small. Our plan for improved energy efficiency is now to simulate this explicitly wasteful algorithm by one in which many of the LISTENs have been replaced by SLEEPs without compromising the correctness of the algorithm, and with a relatively small overhead in time complexity.

Whenever we talk about simulating a radio network algorithm in this paper, we mean replacing it with another equivalent algorithm which is moreover *safe black-box* and *synchronized one-pass*, terms which we are about to define. By equivalent, we mean that at the end of the simulation, each node, v, in the network will have computed its view of a correct transcript of the original algorithm; that is, a record of all messages v sent or received at each timestep in the original algorithm, as well as the correct final internal state. By *black-box*, we mean that when the simulating protocol wants to perform a step of the simulated algorithm, it does so by invoking an oracle that tells it how to update its state, whether to SEND, LISTEN, or SLEEP, and what messages to send. By *safe*, we mean that, the simulating protocol never causes a node to SLEEP during a round in which the original protocol would SEND or receive a message from a SENDing neighbor, regardless of what oracle is provided. Specifically, we don't just mean that the simulating algorithm performs correctly when simulating A; it must avoid SLEEPing incorrectly for all possible simulated algorithms. Finally, by *synchronized one-pass*, we mean that, at certain designated timesteps $\tau(t)$, all nodes either SLEEP or simulate step t of the simulated algorithm. Each timestep is simulated only once, and the timesteps $\tau(t)$ are a strictly increasing function of t. It so happens that for our particular simulation algorithm, the timesteps $\tau(t)$ are a fixed function of t, known in advance, rather than determined adaptively, but this will not be important for our analysis.

4.1 Characterizing Optimal Simulation

In order to come closer to getting our hands on what it would mean for a safe simulation algorithm to be optimal or nearly optimal, we introduce the following generous abstract model for what it needs to do.

Our notion of a generic algorithm simulator is a radio network algorithm, where, at some timesteps, it invokes a black-box simulation of one timestep of the simulated algorithm, A, and at other timesteps, it does its own thing, which may involve metadata gathered from the usage pattern of A. Since A is arbitrary and unknown, we cannot count on knowing the meaning of the messages sent by A, but it may, nevertheless, be useful to know which nodes of A are SENDing, LISTENing, or SLEEPing at a given timestep.

With the above in mind, we define the Generic Simulation Model as follows. For each timestep, at each node, our algorithm must choose to either SEND, LISTEN, SLEEP, or SIMULATE. Steps when our algorithm SENDs, LISTENs or SLEEPS work the same way as in the radio network model. On a SIMULATE step, a black-box for the simulated algorithm, A, simulates one timestep, which may involve SENDING, LISTENING, or SLEEPing. Each node v is given the following information:

1. whether v chose to SEND, LISTEN, or SLEEP in the simulated step.
2. whether any message was received by v in the simulated step.
3. the identity of the next timestep, t_v, at which A would make v SEND, under the assumption that no further messages were received by v before t_v.

At the beginning of the algorithm, we further suppose that the simulator at each node v is given to know the first timestep t_v, at which A would make v SEND, under the assumption that no messages were received by v before t_v. We note that, if the simulated algorithm A were fully event-driven and deterministic, the "unprovoked next-send times" t_v would all be $+\infty$, except for those nodes that send in the first timestep. However, for general algorithms, and in particular, for randomized algorithms, such as the MPX clustering algorithm, there may be many times at which new sequences of SEND steps start without being directly preceded by an incoming message. For randomized algorithms, since we have assumed that each node does all of its coin flipping prior to the beginning of the algorithm, note that the times t_v can be accurately predicted in advance. Since the definition of t_v assumes that no new external information reaches node v between the current step and timestep t_v, the information needed for this calculation is always available to the simulator at the current timestep. Here, we are taking full advantage of the standard assumption that local computation is free; however, we also believe that in most settings, calculating t_v at time t can be done much more quickly than the ultra-naive approach of doing $t_v - t$ steps of black-box simulation of A.

Next, we define a Psychic Synchronized Generic Simulation Model, as follows. Everything about it is the same as for the Generic Simulation Model, above, except that, in addition to being informed about messages received by v in the preceding time step, in this model, we assume that the simulator starts out knowing the entire graph, including node IDs, and is informed, at each time step, of the entire history of metadata at all nodes, including the information about times at which the current node was asleep. In particular, this includes knowing exactly which nodes SENT, LISTENed, and SLEPT at all times $\leq t$, as well as all timesteps $t_{\text{next}}(w, t)$ at which future messages are scheduled to be sent in the absence of provocation, for all vertices $w \in V$.

The other crucial assumption we make in the PSGSM is that all nodes advance their simulation of A in a synchronized way. That is, every node simulates the behavior of A at time t at the same time. This assumption seems natural, considering that the success

or failure of the message deliveries depends on whether collisions occur, but it is still an assumption. Now, considering that in the PSGSM model, all nodes receive, for free, the entire history of relevant metadata for the entire graph, at each timestep they are awake, there seems to be no point in further communication between nodes, except for when steps of A are being simulated. Thus, we find that, in the PSGSM model, the simulated algorithm A should always run in real time.

Now, in both of the above models, we have at least one expectation of a simulation algorithm, and that is *correctness*: at the end of the simulation, we want every node to have its local view of the transcript of the actual computation done by A from the given inputs. In the case of randomized algorithms, we view each node's pre-flipped coins as part of its input, which allows us to reduce to the case of a deterministic algorithm in the usual way.

Our motivation in introducing the Psychic Synchronized (PSGS) model is that, in this model, we can precisely characterize the optimal energy usage of any correct simulation in terms of the behavior of a simple greedy algorithm, Algorithm 1.

Algorithm 1 The Greedy Psychic Algorithm: conserve energy while simulating a given Radio Network agorithm in the PSGSM model.

```
procedure GP(A)                                          ▷ A: the simulated algorithm
    t* ← 1
    for t ← 1 to T do
        if t < t* then
            SLEEP
        else
            SIMULATE timestep t of A
            t* ← t_next(v, t)                            ▷ our next scheduled SEND
            for every vertex w ≠ v do
                psychically receive t_next(w, t)         ▷ time of w's next scheduled SEND
                t* ← min{t*, t_next(w, t) + (dist(v, w) − 1)}
                                                          ▷ SLEEP until time t*
```

▶ **Theorem 11.** *This "Greedy Psychic" algorithm is optimal among all correct simulation algorithms in the PSGSM model, in the sense that, for every simulation algorithm SIM in this model, either there exists a radio network algorithm A and an input x such that SIM(A, x) makes a mistake (some node does not end with the correct transcript), or, for every algorithm A, input x, and vertex v, we have* $\mathrm{ENERGY}(GP, A, x, v) \leq \mathrm{ENERGY}(\mathrm{SIM}, A, x, v)$.

The main consequence of Theorem 11 will be that Algorithm SAF is within a polylog(n) multiplicative factor of optimal, at least among algorithms that perform a single synchronized simulation of the simulated algorithm A. This near-optimality holds in a vertex-by-vertex and algorithm-by-algorithm manner. We point out that without the assumption of synchronicity, we cannot expect to get guarantees of this kind. For example, if we want to make an asynchronous algorithm that minimizes the listening cost for a specific vertex v, we can make v wake up very infrequently, and ensure that its neighbors are always "holding v's messages ready for it." In this way, the energy complexity for v can be reduced essentially to the number of messages that v needs to send or receive; this favoritism towards v would presumably be more than paid for by increased costs incurred at other vertices. This strongly suggests that, without the assumption of synchronous one-pass simulation, there may not exist a single simulation algorithm that simultaneously minimizes the energy costs for all nodes.

5 The Simulation Algorithm

In this section we describe our recursive simulation algorithm. We begin with a section describing the underlying assumptions, and then give a high-level overview of the algorithm. The actual pseudocode appears in Section 5.3. We analyze the the algorithm and its energy complexity in Section 5.4.

5.1 Assumptions

We assume that we are given a function f describing protocol to be run on G, that is guaranteed to succeed in the OR model of message delivery. Recall that, as mentioned in Section 2, we may assume f is deterministic without loss of generality, since each node may do any necessary coin flipping in advance, at the beginning of the algorithm. Note that, since nodes are not privy to each others inputs, this assumption does not imply that randomness is shared. In particular, if a node v were to randomly choose a timestep at which to SEND a message, the other nodes would not automatically know which timestep it was; however, v would know in advance, and the Simulation algorithm would have v warn its neighbors.

We also assume that, prior to attempting to execute our algorithms indexed by j, at least levels $1, 2, \ldots, j$ of a (R, α, β) multi-scale clustering have been computed. Each node knows the ID of its cluster center, as well as its graphical distance to the cluster center. (This information is used in UPCAST/DOWNCAST.)

Our algorithms all have the property that each call to one of the defining functions takes the same number of timesteps, including those spent in recursive calls. In some cases, this number of timesteps can be tedious to compute; therefore, we have taken the liberty to refer to the number of "simulated" timesteps, namely the number of timesteps of the simulated algorithm. This should be understood as a shorthand only. Here is a complete list of the state information that each node must store in its memory.

- Shared knowledge of the graph parameters n, Δ, D. Here, n is an upper bound on the number of nodes in G, Δ is an upper bound on the maximum degree of G, D is an upper bound on the diameter of D. If Δ and D are not specified, $n-1$ can be used in their place, (although better bounds lead to better performance). We require that all nodes in the network start out initialized with the same values of these parameters.
- The current timestep, t. Since we are assuming a synchronous network, this value is always the same for all nodes.
- The cluster parameters R_j, α_j, β_j, for $1 \leq j \leq \ell$. This knowledge is also shared by all nodes in the network.
- A unique ID for our node. If these are not specified, a random string of $C \log n$ bits may be used; by the birthday paradox, these are distinct with high probability.
- For $1 \leq j \leq \ell$, the ID of our level-j cluster center, as well as our graphical distance to the cluster center. This is used for energy-efficient communication by the UPCAST/DOWNCAST subroutines (see [7]).
- A message string, m. Initially null for all nodes. This is used in the NOTIFY protocol.
- Any state information stored by our node in the simulated protocol, f.

5.2 Algorithm Overview

At a high level, the SAF simulation algorithm gets an algorithm f and a time interval I, and has the goal of simulating a run of f on I while allowing processors that are far from the action in f to save energy by sleeping until the appropriate time. To this end, SAF

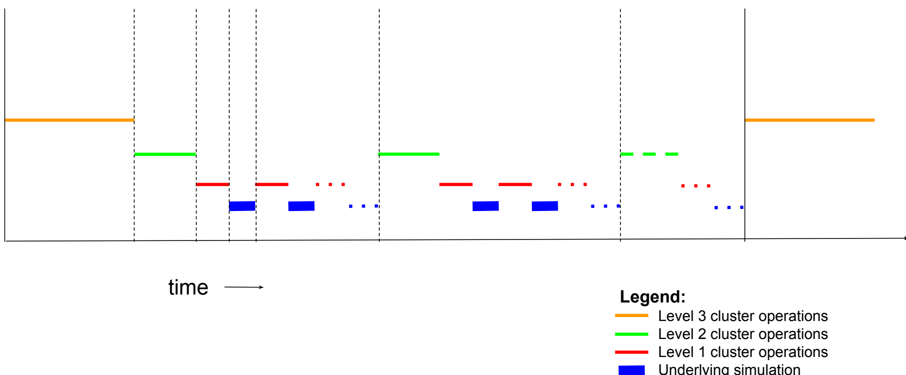


Figure 1 Timeline of how cluster operations at various levels interleave with the underlying simulated algorithm. Each node starts at the top level (here $\ell = 3$) and participates in cluster operations until it finds a level at which it can drop out. That level also determines when it will wake up next: the next round of cluster operations at that level. If the node gets to the bottom level of cluster operations without dropping out, then it stays awake for the underlying simulation for the next few simulated timesteps.

simulation uses a multi-scale clustering to estimate the distance to the nodes where the communication is happening, and to set SLEEP and WAKE schedules for the nodes that need to wait. The main idea is that the clusters at level j, which are (R_j, α_j, β_j) approximately distance-preserving, are equipped to make a decision (SLEEP/WAKE) from their cluster that is pertinent to the next R_j steps of the simulated algorithm. This decision is made for each cluster, collectively by its members, by performing internal cluster operations to decide whether there is activity nearby. However, since their confidence in the prediction of inactivity is only good for R_j simulated timesteps, this would appear to mean that they must wake up and perform cluster operations after every R_j steps of the simulated algorithm. When R_j is small, this would not actually help them save much energy. However, this is where the clusterings at multiple scales come in. At the highest scale, ℓ, the clusters wake up every R_ℓ steps, so as long as the total simulated time that f needs to run is at most $R_\ell \text{polylog} n$, the algorithm is in good shape. The notion of what it means for the action to be far away has been tuned so that nesting property described in Lemma 9 ensures that if a top level cluster has gone to sleep, then all the lower level clusters that are contained within can also afford to sleep for the full time horizon of R_ℓ steps. If a top level cluster decides to stay awake, then it sets up a recursive call to enable affected lower level clusters to decide how many times they must wake up with the shortened time horizon. Note that the top level cluster staying awake does not mean that an individual node within that cluster must sat awake for R_ℓ timesteps. Indeed, that would expend far too much energy. Rather, it means that such a node must participate in lower level cluster operations, until it can find a scale at which the computational activity of f is far away from it. Only if a node finds itself to be awake at *all* scales will it participate in the actual simulation of f, for the next interval of the smallest scale length, R_1.

But how to get the multi-scale clustering to begin with? Herein lies the beauty of the process. At the bottom level, an (R_1, α_1, β_1) approximate distance-preserving partition can be computed with polylog(n) energy and polylog(n) latency, with all processors staying awake the whole time. (We need R_1 to be polylog(n).) Thereafter, the simulation has a multi-scale clustering to work with, with $\ell = 1$ the first time around; and the partitioning algorithm is itself a function that is suitable for being simulated recursively. Thus, each subsequent level of the final multi-scale clustering is bootstrapped off the previous ones.

5.3 Pseudocode

Our code is divided into several pieces, which are presented as Algorithms 2 through 5. The SAF Simulation algorithm is stated in a form where it can be applied using an arbitrary multi-scale clustering. In general, we want to apply it to the multi-scale clustering we know how to construct for general graphs, namely the ESTCluster algorithm of Miller, Peng, and Xu. To build this clustering efficiently, we simulate it using the SAF algorithm inductively. Finally, to efficiently run BFS from an arbitrary start node, we apply SAF Simulation to the naive parallel BFS algorithm that has each node listen until the BFS frontier reaches it, then send a message to its neighbors to advance the frontier while incrementing its level counter.

Algorithm 2 SAF Simulation Algorithm: Adaptively power down receiver to save energy while simulating a given algorithm, f. Assumption: we have a hierarchy of α, β, R.

```
procedure SAF(j, I, f)    ▷ j: current cluster height, I: interval of simulated times, f:
update rule for the simulated algorithm (on level 0 clusters)
    if j = 0 then                                          ▷ Bottom level (single node)
        (Naively) execute f for |I| timesteps.
    else
        Partition I into disjoint subintervals, I_1, I_2, ..., I_k, each of length R_j.
        for i ← 1 to k do
            Internally simulate f on I_i assuming no messages received.
            m ← { null                  if we never sent during the simulation.
                { "Activity nearby!"    if we sent anything during the simulation.
            if NOTIFY(j, m, 2α_j) returns "OK TO SLEEP" then
                SLEEP until end of last simulated timestep in I_i.
                Update state based on the above simulation.
            else
                SAF(j − 1, I_i, f)
```

5.4 Formal Statements of the Results

▶ **Definition 12.** *Suppose P is an ℓ-level multiscale clustering with parameters $(\mathbf{R}, \boldsymbol{\alpha}, \boldsymbol{\beta})$. Then, for $0 \le j \le \ell$, we define a* level-j epoch *to be any time interval of the form*

$$\{iR_j + 1, iR_j + 2, \ldots, (i+1)R_j\}.$$

We will use the following result relating distances from the active set to the event of being awake and made to participate in the NOTIFY subroutine.

Algorithm 3 Ensure that all nearby clusters (of a given height) wake up when something interesting is happening nearby.

```
procedure NOTIFY(j, m, r)    ▷ j: current cluster height, m: message to send, r: transmission radius ▷ If m = null, do not initiate sending, but do repeat any message you hear.
    for i ← 1 to r do
        UPCAST(j,m)
        DOWNCAST(j,m)
        INTERCAST(j,m)
    UPCAST(j,m)
    DOWNCAST(j,m)
    if we received or sent any messages during this call to NOTIFY then
        return "STAY AWAKE"
    else
        return "OK TO SLEEP"
```

Algorithm 4 Naively build MPX clusters with radius parameter R.

```
procedure NAIVELY-BUILD-MPX(R)    ▷ R: radius parameter
    my_cluster_center[j] ←null
    Sample a weight W from the exponential distribution with mean R.
    t_max ← ⌈3R log(n)⌉
    for i ← 1 to t_max do
        if my_cluster_center[j] ≠ null then
            SEND message (my_cluster_center[j],my_cluster_depth[j]).
            Break out of FOR loop, and SLEEP for remaining t_max − i timesteps.
        else if i + W ≥ t_max then
            my_cluster_center[j] ←my_ID
            my_cluster_depth[j] ← 0
        else
            LISTEN this timestep.
            if message (c, d) received then
                my_cluster_center[j] ← c
                my_cluster_depth[j] ← d + 1
```

Algorithm 5 Efficiently build a geometric multiscale clustering with parameter $O(\log^2 n)$.

```
procedure BUILD-MSC
    R ← C log^2(n)
    for j ← 1 to log_R(n) do
        SAF(j − 1, [0, ⌈3R^j log(n)⌉], Naively-Build-MPX(R^j))
```

▶ **Lemma 13.** *For $j \geq 1$, $t = (i+1)R_j$, $v \in V$, if the distance from v to the nearest node that would send in the epoch ending at time t is at least $(2\alpha_j + 1)(\beta_j R_j + 1)$, then, v will not participate in any calls to NOTIFY($j-1$) within this epoch.*

Now we are ready to state our main result about our simulation algorithm.

▶ **Theorem 14.** *Let f be any randomized radio network protocol in the OR model. Suppose P is an ℓ-level multiscale clustering with parameters $(\mathbf{R}, \boldsymbol{\alpha}, \boldsymbol{\beta})$. Then SAF($\ell, I, f$) has the following properties:*

- $\mathbf{P}$ *(SAF(ℓ, I, f) succeeds)* $\geq \mathbf{P}$ *(f succeeds)* $- 1/\text{poly}(n)$.
- *The running time of SAF(ℓ, I, f) is T_f polylog(n), where T_f is the running time of f.*
- *The energy cost of SAF(ℓ, I, f) for a vertex v is at most $\mathcal{E}$ polylog$(n) + T \log n / R^\ell$, where $\mathcal{E}$ is the energy cost of the greedy psychic algorithm $GP(f)$ for vertex v, and $T = \text{TIME}(f)$ is the time complexity of the simulated algorithm.*

▶ **Theorem 15.** *The Build-MSC algorithm (Algorithm 5) builds a multi-scale clustering at all scales, with probability $1 - 1/\text{poly}(n)$, with total running time $O(D\ \text{polylog}(n))$ and per-node energy usage $O(\text{polylog}(n))$.*

6 BFS Revisited

In this section we apply our methodology to get a polylog energy algorithm for Breadth First Search in radio networks, thus answering an open question from [8]

The algorithm is very simple: we simply simulate the naive BFS algorithm for radio networks in our SAF simulation framework.

Algorithm 6 Solve BFS from a designated vertex, v in low energy. D is an upper bound on the diameter.

```
procedure Efficient-BFS(v)
    Build-MSC
    SAF(ℓ, [0, D], Naive-Parallel-BFS(R^j))
```

▶ **Theorem 16.** *The Efficient-BFS algorithm (Algorithm 6) computes the graphical distance to each node from the root vertex, with probability $1 - 1/\text{poly}(n)$. Its total running time is $O(D$ polylog$(n))$ and per-node energy usage is $O(\text{polylog}(n))$.*

7 Conclusion

We have shown a new general-purpose methodology for reducing the energy cost of Radio Network algorithms by collaborating with clusters of nearby nodes at multiple scales to detect when it is safe to shut off the receiver due to there being no danger of message activity nearby. Although similar techniques have been used in previous work, the precise way in which we create and use our clusters leads, at least in some cases, to significantly improved results. In particular, our methodology allows us to easily, and at least in the case of BFS, significantly improve over known results.

References

1 R. Bar-Yehuda, O. Goldreich, and A. Itai. On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. *Journal of Computer and System Sciences*, 45(1):104–126, 1992.

2 M. Barnes, C. Conway, J. Mathews, and D. K. Arvind. ENS: An energy harvesting wireless sensor network platform. In *Proceedings of the 5th International Conference on Systems and Networks Communications (ICSNC)*, pages 83–87, 2010.

3 M. Bender, T. Kopelowitz, S. Pettie, and M. Young. Contention resolution with constant throughput and log-logstar channel accesses. *SIAM J. Comput.*, 47:1735–1754, 2018.

4 P. Berenbrink, C. Cooper, and Z. Hu. Energy efficient randomised communication in unknown adhoc networks. *Theoretical Computer Science*, 410(27):2549–2561, 2009.

5 Y.-J. Chang, T. Kopelowitz, S. Pettie, R. Wang, and W. Zhan. Exponential separations in the energy complexity of leader election. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 771–783, 2017.

6 Yi-Jun Chang. Energy complexity of distance computation in multi-hop networks. *CoRR*, abs/1805.04071, 2018. arXiv:1805.04071.

7 Yi-Jun Chang, Varsha Dani, Thomas P Hayes, Qizheng He, Wenzheng Li, and Seth Pettie. The energy complexity of broadcast. In *Proceedings of the 37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 95–104, 2018.

8 Yi-Jun Chang, Varsha Dani, Thomas P Hayes, and Seth Pettie. The energy complexity of BFS in radio networks. In *Proceedings of the 39th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 273–282, 2020.

9 Soumyottam Chatterjee, Robert Gmyr, and Gopal Pandurangan. Sleeping is efficient: Mis in o (1)-rounds node-averaged awake complexity. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 99–108, 2020.

10 I. Chlamtac and S. Kutten. On broadcasting in radio networks-problem analysis and protocol design. *IEEE Transactions on Communications*, 33(12):1240–1246, 1985.

11 Varsha Dani, Aayush Gupta, Thomas P. Hayes, and Seth Pettie. Wake up and join me! an energy-efficient algorithm for maximal matching in radio networks. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)*, volume 209 of *LIPIcs*, pages 19:1–19:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.DISC.2021.19.

12 L. Gasieniec, E. Kantor, D. R. Kowalski, D. Peleg, and C. Su. Energy and time efficient broadcasting in known topology radio networks. In *Proceedings 21st International Symposium on Distributed Computing (DISC)*, pages 253–267, 2007.

13 S. Gilbert, V. King, S. Pettie, E. Porat, J. Saia, and M. Young. (Near) optimal resource-competitive broadcast with jamming. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 257–266, 2014.

14 B. Haeupler and D. Wajc. A faster distributed radio broadcast primitive. In *Proceedings of the 35th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 361–370, 2016.

15 T. Jurdzinski, M. Kutylowski, and J. Zatopianski. Efficient algorithms for leader election in radio networks. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 51–57, 2002.

16 T. Jurdzinski, M. Kutylowski, and J. Zatopianski. Energy-efficient size approximation of radio networks with no collision detection. In *Proceedings of the 8th Annual International Conference on Computing and Combinatorics (COCOON)*, pages 279–289, 2002.

17 T. Jurdzinski, M. Kutylowski, and J. Zatopianski. Weak communication in radio networks. In *Proceedings of the 8th International European Conference on Parallel Computing (Euro-Par)*, pages 965–972, 2002.

18 T. Jurdzinski, M. Kutylowski, and J. Zatopianski. Weak communication in single-hop radio networks: adjusting algorithms to industrial standards. *Concurrency and Computation: Practice and Experience*, 15(11–12):1117–1131, 2003.

19 T. Jurdzinski and G. Stachowiak. Probabilistic algorithms for the wakeup problem in single-hop radio networks. In *Proceedings of the 13th International Symposium on Algorithms and Computation (ISAAC)*, pages 535–549, 2002.

20 J. Kabarowski, M. Kutylowski, and W. Rutkowski. Adversary immune size approximation of single-hop radio networks. In *Proceedings Third International Conference on Theory and Applications of Models of Computation (TAMC)*, pages 148–158, 2006.

21 M. Kardas, M. Klonowski, and D. Pajak. Energy-efficient leader election protocols for single-hop radio networks. In *Proceedings 42nd International Conference on Parallel Processing (ICPP)*, pages 399–408, 2013.

22 V. King, S. Pettie, J. Saia, and M. Young. A resource-competitive jamming defense. *Distributed Computing*, 31:419–439, 2018.

23 M. Klonowski and D. Pajak. Brief announcement: Broadcast in radio networks, time vs. energy tradeoffs. In *Proceedings 37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 115–117, 2018. doi:10.1145/3212734.3212786.

24 Marek Klonowski and Malgorzata Sulkowska. Energy-optimal algorithms for computing aggregative functions in random networks. *Discrete Mathematics & Theoretical Computer Science*, 17(3):285–306, 2016.

25 M. Kutylowski and W. Rutkowski. Adversary immune leader election in ad hoc radio networks. In *Proceedings 11th Annual European Symposium on Algorithms (ESA)*, pages 397–408, 2003. doi:10.1007/978-3-540-39658-1_37.

26 G. L. Miller, R. Peng, and S. C. Xu. Parallel graph decompositions using random shifts. In *Proceedings of the 25th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 196–203, 2013.

27 Gary L Miller, Richard Peng, Adrian Vladu, and Shen Chen Xu. Improved parallel algorithms for spanners and hopsets. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 192–201, 2015.

28 K. Nakano and S. Olariu. Energy-efficient initialization protocols for single-hop radio networks with no collision detection. *IEEE Trans. Parallel Distrib. Syst.*, 11(8):851–863, 2000.

29 Daisy Phillips. Tessellation. *Wiley Interdisciplinary Reviews: Computational Statistics*, 6(3):202–209, 2014.

30 J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN)*, pages 364–369, 2005.

A Proofs

Proof of Lemma 3. Let $u, v \in V$. Let $d^*([u],[v]) = k$ and let $W_0, W_1, \ldots, W_k \in V(G/\mathcal{P})$ be such that $W_0 = [u], W_k = [v]$ and $W_0, W_1, \ldots, W_k$ is a shortest path from $[u]$ to $[v]$ in $G/\mathcal{P}$. Then, for $0 \le i < k$, since W_i and W_{i+1} are adjacent in $G/\mathcal{P}$, there are vertices $y_i \in W_i, x_{i+1} \in W_{i+1}$ such that $(x_{i+1}, y_i) \in E(G)$. Let $x_0 = u$ and $y_k = v$. For all i, since x_i and y_i are in the same cluster, $d(x_i, y_i) < \beta R$, and if we connect these pairs by shortest paths, we have constructed a path of length at most $k + (k+1)\beta R$ from u to v in G. It follows that

$$d(u,v) \le k + (k+1)\beta R = k(\beta R + 1) + \beta R$$

Since $k = d^*([u],[v])$, rearranging terms gives us the first inequality.

Now suppose $d(u,v) = qR + r$, where $r < R$. Consider a shortest path from u to v in G, and let w_i be the vertices at distance iR from u on the path. The path has thus been cut into $q' = q + \lfloor r/R \rfloor = \lceil d(u,v)/R \rceil$ pieces each of length at most R. Let $w_{q'} = v$. By the first property, $d^*([w_i],[w_{i+1}]) \le \alpha$. Therefore, by the triangle inequality,

$$d^*([u],[v]) \le \alpha \left\lceil \frac{d(u,v)}{R} \right\rceil$$

which completes the proof. ◀

Proof of Lemma 4. If X is an exponentially distributed random variable with mean R then $\mathbf{P}(X > t) = e^{-t/R}$. Thus, the probability that $X > 3R\log n$ is at most $1/n^3$. By a union bound, it follows that with probability at least $1 - \frac{1}{n^2}$ the first cluster starts forming *after* time $-3R\log n$ and since each vertex either joins a cluster or starts its own by time $t = 0$, the cluster diameters are at most $3R\log n$. ◀

Proof of Lemma 5. This is just a restatement in our notation of Lemma 2.1 from [8], and is proved there. ◀

Proof of Corollary 6. Setting $\ell = R$ we see that for any $v \in V$, the probability that there are more than $C\log n$ clusters intersecting the ball of radius R around v is at most

$$\left(1 - \frac{1}{e^2}\right)^{C\log n} \leq \exp\left(\frac{-C\log n}{10}\right).$$

The result follows. ◀

Proof of Lemma 9. Let $w \in B_j(v, 2\alpha_j)$. Then, by definition, $d_j([v]_j[w]_j) \leq 2\alpha_j$. Since $\mathcal{P}_j$ is an (R_j, α_j, β_j) approximate distance-preserving partition, by Lemma 3,

$$\left\lfloor \frac{d(v,w)}{\beta_j R_j + 1} \right\rfloor \leq 2\alpha_j$$

Removing the floor and rearranging the terms, we get

$$\begin{aligned} d(v,w) &\leq (\beta_j R_j + 1)(2\alpha_j + 1) \\ &\leq (2\alpha_j + 1)(\beta_j + 1)R_j \\ &\leq R_{j+1} + R_j \end{aligned}$$

where the last line follows from Equation (1). By the triangle inequality,

$$d(u,w) \leq d(u,v) + d(v,w) \leq R_{j+1} - R_j + R_{j+1} + R_j = 2R_{j+1}$$

Applying Lemma 3 again, we have

$$d_{j+1}([u]_{j+1}, [v]_{j+1}) \leq \alpha_{j+1} \left\lceil \frac{d(u,v)}{R_{j+1}} \right\rceil \leq 2\alpha_{j+1}$$

so that $w \in B_{j+1}(u, 2\alpha_{j+1})$, completing the proof. ◀

Proof of Theorem 11. The proof is by induction of the number of computational steps. Suppose we have fixed A, v, x, and that SIM is known to be an always-correct simulator. Our inductive hypothesis is that, by the times the cumulative energy costs of $Greedy(A, v, x)$ and $\text{SIM}(A, v, x)$ both reach i, the greedy simulation will have completed simulating at least as many timesteps of A as SIM will have. Assume the hypothesis for $i - 1$. Then SIM wakes up for the i'th time at or before Greedy does, say at time t. At this point it can go back to sleep, but the latest it can sleep is $t^*(v, t)$, defined as above, since SIM needs to be safe. Greedy wakes up for the i'th time at some time $t' \geq t$, and goes to sleep until time exactly $t^*(v, t')$. Now, the crucial point is that the function $t^*(v, \cdot)$ is an increasing function of its second argument, which is clear from its intention, but also from the facts that $t_{\text{next}}(w, \cdot)$ can only decrease when w receives a message from a neighbor, which implies that $t^*(w, t) - t \leq 1$, and that $\mathsf{dist}(w, \text{SENDERS})$ never decreases by more than 1 in a single timestep. Hence $t^*(v, t') \geq t^*(v, t) \geq$ whenever SIM decides to wake back up, which completes the inductive step. ◀

Proof of Lemma 13. Let u be the nearest node to v that would initiate a SEND in the epoch. In order to participate in a call to NOTIFY$(j-1)$, v must receive a "WAKE UP" return value from the call to NOTIFY(j) at the beginning of the epoch. However, this will not happen because, by Lemma 3, $d^*([u],[v]) \geq \left\lfloor \frac{d(u,v)}{\beta_j R_j+1} \right\rfloor \geq 2\alpha_j$, but this means, if v participates in the NOTIFY, it will instead get a return value of "OK TO SLEEP." Here d denotes distance in the underlying graph, and d^* denotes distance in the level-j cluster graph. ◀

Proof of Theorem 14. First, observe that, after each NOTIFY operation, a node v stays awake if and only if it is within $2\alpha_j$ level-j clusters of the set of active nodes for the beginning of the corresponding level-j epoch. That is, assuming that all of our cluster operations (Upcast/Downcast/Intercast) succeed. Since, with high probabiltity, this happens at all timesteps, let us discount the $1/\text{poly}(n)$ chance of a failure.

Since the level-j epoch is R^j time units long, every active node in the entire epoch is within distance R^j of the initial set of active nodes. By the definition of approximately distance-preserving partition, these nodes are all within distance α_j of the initial active set in the level-j cluster graph. Since we doubled this radius of notification in the cluster graph, it follows by Lemma 9 that all nodes in lower-level clusters that wake up during the simulation of this level-j epoch are already awake at level j, and therefore their lower level cluster operations will succeed.

In particular, at level 0, it follows that any time protocol f has a node Send, and one of its neighbors Listen, that both nodes in question will be awake and simulating f at the corresponding timestep. It follows by induction that the state of all processors at any simulated time t is consistent with f run under the OR model until time t. Hence, the final outcome will also be consistent with f run under the OR model.

For the latency analysis, we observe that the number of level-j epochs is T_f/R^j where T_f is the running time of f. For each such epoch, we do $O(\alpha_j)$ level-j cluster operations (Upcast/Downcast/Intercast), which each require time $O(\beta_j R_j \log(n) \log\log(n))$, since the clusters have diameter at most $\beta_j R_j$, and the backoff requires $\log(\log(n))$ time because each node is, with high probability next to $O(\log n)$ different clusters; the further $O(\log(n))$ factor is to guarantee success with high probability. Summing over all epochs, we get a total running time of $O(T_f \alpha_j \beta_j \log(n) \log\log(n))$ attributable to level-j operations. Finally, summing over the $\leq \log(n)$ levels, and using that $\alpha_j, \beta_j = O(\log n)$ for all j, we get a total running time of $O(T_f \log^4(n) \log(\log(n)))$.

For the energy analysis, we consider the time interval between two consecutive non-SLEEP actions by the Greedy Psychic algorithm, so that OPT is spending one energy. If this interval has length L, then we know that after the first i timesteps of this interval, the distance from v to the nearest active vertex is always at least $L-i$. Consequently, applying Lemma 13, we know that v can, at most, participate in a subset of the final $(2\alpha_j+1)\beta_j$ of the calls to NOTIFY(j) during this time interval, assuming $j < \ell$. Since each call to NOTIFY costs at most $O(\alpha_j \log n)$ energy per participating vertex, this means our algorithm spends at most $O(\ell\alpha_j \log n)$ times more energy than the Greedy Psychic algorithm does. Finally, for the top level, we get T/R_ℓ calls to NOTIFY(ℓ) in all, each at a cost of $\alpha_j \log n$, regardless of distances. Summing these energy costs completes the proof. ◀

Proof of Theorem 15. By Theorem 14, we know that the algorithm succeeds in building each level of the clustering with essentially the same success probability as Naive-Build-MPX (Algorithm 4). Since the naive algorithm runs in time proportional to the radius parameter, and these radius parameters form a geometric sequence from 1 up to $O(D)$, the total running

time for all the calls to Naive-Build-MPX is $O(D\text{polylog}(n))$. Similarly, we can estimate the expected number of active j'-epochs for Naive-Build-MPX(R^j) as $O(R)$. This is because the active set moves out from the cluster centers at unit velocity, so once it gets within distance $R^{j'}$ of a vertex, all activity within distance $R^{j'}$ ceases within at most $2R^{j'}$ timesteps, by the Triangle Inequality. Thus the total number of active j-epochs for a given node is $O(1)$. Summing over all $0 \le j \le \ell$, we get a total energy use of $\tilde{O}(\ell)$, which is polylog(n). ◀

Proof of Theorem 16. Theorem 15 gives us the running time and cost for the cluster formation. Then we can use Theorem 14 to analyze the SAF simulation of the naive BFS algorithm. The analysis of the number of active epochs here is essentially the same as in the proof of Theorem 15, since the active set again passes very quickly through each region that it enters. We leave the details to the reader. ◀

B Simulating cluster-graph algorithms on the underlying graph

Suppose we already have approximately distance-preserving partitions at all scales from 1 up to the diameter of G. The ability to run graph algorithms on the corresponding quotient graphs will be a rather useful primitive to add to our toolbox. Because the distances in these graphs are scaled down by a large factor, we may reasonably expect that they can be run at a much lower cost. But what is the cost of simulating these algorithms on the actual network?

For the most part, this question was already answered by Chang, Dani, Hayes and Pettie [7]. We briefly describe the approach used to simulate one timestep of computation on the cluster graph. Each node within one cluster in our partition uses part of its memory to record the state of a hypothetical processor corresponding to the cluster. At the beginning and end of the simulated timestep, we require that this state be the same for every node in the cluster. Depending on whether the cluster state indicates we should Send, Listen, or Sleep, every node in the cluster does this (INTERCAST). Next, if the operation was Listen, all the nodes that received a message propagate these up towards the cluster center (UPCAST). Since every node in the cluster knows its distance from the root, this propagation can be synchronized so that each node only needs to Listen once, in the same timestep that its children might Send. Since we are in the OR model, we only require that, from among the messages received, an arbitrary one is received. Next, the cluster center updates its state based on the received message, and broadcasts the result within the cluster (DOWNCAST), after which the simulation of one computational step on the cluster graph is complete.

Algorithm 7 High-level description of Upcast, Downcast, and Intercast algorithms, from [7].

▷ For each of the subroutines below, we require that either all the nodes in a particular cluster participate, or none do.

procedure UPCAST(j, m) ▷ j: current cluster height, m: message to send

 ▷ Guarantee: if any node in the cluster participates with a non-null message, then one of these messages is received by the cluster center, who then stores it in their message variable.

procedure DOWNCAST(j, m) ▷ j: current cluster height, m: message to send

 ▷ Guarantee: if the cluster center participates with a non-null message, then this message is received by each node in the cluster, who then stores it in their message variable.

procedure INTERCAST(j, m) ▷ j: current cluster height, m: message to send

 ▷ Guarantee: if at least one neighboring node is contained in a participating level-j cluster that has a non-null message, then this node receives such a message, and store it in their message variable.

Since the messages are being broadcast, rather than sent along edges, there are some subtleties in the details. For instance, even though we have described the algorithm under the assumption that message delivery in both the underlying graph, and in the cluster graph, takes place using the collision-free OR model, there is still a need for backoff, to prevent messages accidentally crossing between adjacent clusters during UPCAST and DOWNCAST. However, for purposes of the present work, these details are unimportant.

Since each of the three stages, INTERCAST, UPCAST, and DOWNCAST, costs $\tilde{O}(1)$ energy per node in the cluster, the per-node energy cost for running an algorithm on the cluster graph is within a polylog factor of the per-node energy cost for simulating it on the underlying graph. When the cluster "radius" parameter is R, the latency for the UPCAST and DOWNCAST is $\tilde{O}(R)$, so this becomes a multiplicative factor for the time complexity.

Contention Resolution Without Collision Detection: Constant Throughput *And* Logarithmic Energy

Gianluca De Marco ✉
Department of Computer Science, University of Salerno, Italy

Dariusz R. Kowalski ✉
School of Computer and Cyber Sciences, Augusta University, GA, USA

Grzegorz Stachowiak ✉
Institute of Computer Science, University of Wrocław, Poland

Abstract

A shared channel, also called a multiple access channel, is among the most popular and widely studied models of communication in distributed computing. An unknown number of stations (potentially unbounded) is connected to the channel and can communicate by transmitting and listening. A message is successfully transmitted on the channel if and only if there is a unique transmitter at that time; otherwise the message collides with some other transmission and nothing is sensed by the participating stations. We consider the general framework without collision detection and in which any participating station can join the channel at any moment. The contention resolution task is to let each of the contending stations to broadcast successfully its message on the channel.

In this setting we present the first algorithm which exhibits asymptotically optimal $\Theta(1)$ throughput and only an $O(\log k)$ energy cost, understood as the maximum number of transmissions performed by a single station (where k is the number of participating stations, initially unknown). We also show that such efficiency cannot be reproduced by non-adaptive algorithms, *i.e.*, whose behavior does not depend on the channel history (for example, classic backoff protocols). Namely, we show that non-adaptive algorithms cannot simultaneously achieve throughput $\Omega\left(\frac{1}{\text{polylog}(k)}\right)$ and energy $O\left(\frac{\log^2 k}{(\log\log k)^2}\right)$.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Shared channel, Contention resolution, Throughput, Energy consumption, Randomized algorithms, Lower bound

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.17

Funding *Dariusz R. Kowalski*: partially supported by the National Science Foundation grant No. 2131538 and the Polish National Science Center (NCN) grant UMO-2017/25/B/ST6/02553.

1 Introduction

A shared channel, or a multiple access channel, is one of the fundamental communication models: it allows many autonomous computing entities to communicate over a shared medium and the main challenge is how to efficiently resolve collisions occurring when more than one entity attempts to access the channel at the same time (*c.f.*, the surveys by Gallager [18] and Chlebus [7]).

© Gianluca De Marco, Dariusz R. Kowalski, and Grzegorz Stachowiak;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 17; pp. 17:1–17:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper we consider a classical setting, formally described as follows (c.f., [6]).[1] A *potentially unbounded* number of stations is attached to a shared channel, each of them possessing a packet that can be transmitted in a single time slot. The stations are anonymous in that they do not have any identification label (ID).[2] The computation is decentralized: every station acts independently by means of its own distributed protocol.

The communication proceeds in synchronous rounds. In each round a station can either transmit a message or listen. A transmission is successful in a given round if and only if exactly one station is transmitting in this round. In this case, the message is delivered to all stations currently active on the channel. If more than one station transmits in the same round we say that a *collision* occurs: the transmitted messages interfere each other and retransmission is necessary. Messages could be either original packets or constant-size control messages.[3]

Contention resolution problem. Each station wakes up at arbitrary time with a single packet, and in each computation prefix the number of awaken stations is finite; we will often denote it by parameter k. A station may terminate and leave the system (switches off) after its packet has been successfully transmitted. When a station switches off, it disconnects permanently from the channel. A station already awaken but not switched off yet, will be called *active*.

A *contention resolution* algorithm is a distributed algorithm that schedules the transmissions for each participating station, guaranteeing that every active station eventually transmits successfully. In other words, for each station the algorithm has to guarantee that there exists a round at which that station transmits individually, *i.e.*, without interfering with other stations.

No collision detection. Our setting is *without collision detection*. This means that in case of collision the stations do not perceive any special signal, so it will be impossible for them to distinguish between the case when more stations transmit simultaneously and the case of a silent round where no station transmits. The only feedback an active station can sense is when some station itself transmits successfully, in which case all active stations receive the transmitted message. This assumption, also called a (system) acknowledgement, is quite common in the literature (see *e.g.* [1, 3, 2]), and well motivated by technological applications such as CSMA/CA [4].

Dynamic scenario. The literature on the contention resolution problem started in the 70's and mostly considered the (simplified) *static* situation in which the stations are all activated at the same time, and thus start simultaneously their protocols, or that the activation times are based on statistical or adversarial-queueing models.

Continuing on a more recent line of research (*cf* [28, 17, 2, 3, 37, 36, 36, 35, 12, 11, 34]), in this paper we consider a general and realistic *dynamic* scenario, in which the stations get awake at arbitrary times and the sequence of activation times is totally determined by a worst-case adaptive adversary. Since a station can start its local execution of the protocol only after it has been woken up, there is no synchronization among the protocols. This makes the problem of designing a distributed algorithm considerably more challenging with respect to its simplified static counterpart.

---

[1] This setting was also used for modelling and analysis of CSMA/CA technology, c.f., [4].
[2] We assign names to the stations only for the purpose of distinguishing them in the analysis of correctness and performance.
[3] CSMA/CA and many other wireless technologies assume small-size control messages.

Local clock. Although the communication proceeds in synchronous rounds with the clocks of all the stations ticking at the same rate, our model does not allow any global clock. Each station can measure time only on the basis of its own *local clock*, which starts in the round at which the station wakes up and therefore, in the dynamic scenario considered in this paper, is not synchronized with the other clocks. We conventionally assume that a station is activated at round 0 of its local clock and can start transmitting since round 1.

Algorithmic solutions. Since the stations are anonymous, they are identical from a deterministic perspective; consequently, the only feasible solutions must be randomized. We consider *randomized* distributed algorithms for the contention resolution problem: every woken-up station has to transmit successfully its packet, regardless of its (adversarial) activation time.

All our asymptotic bounds are to be understood as high probability bounds, that is, they hold *with high probability* (in short: *whp*). We say that an event for an algorithm holds whp(k), when for a predefined parameter $\eta > 0$, the parameters of the algorithm can be chosen, so that for any k the event holds with probability at least $1 - 1/k^\eta$. In the intermediate steps of analysis, we will sometimes need to use the notion of whp not only with respect to the pre-assumed parameter η; in such case we will say more specifically that an event occurs "whp $1 - 1/k^\lambda$", for some $\lambda > 0$. Parameter λ will typically be slightly higher than η, so that at the end we could get the final result with the sought probability at least $1 - 1/k^\eta$.

Complexity measures. In this paper we measure the efficiency of the algorithms both in terms of *throughput* and *energy consumption*. The *throughput* is defined as follows. Given any time $t > 0$, if $n[t]$ denotes the number of stations activated up to t and $r[t]$ the number of *active rounds*, that is the rounds at which at least one station is still in the system (*i.e.,* not yet switched-off as a result of successful transmission), then the throughput is defined as the ratio $n[t]/r[t]$. Again, the goal of an algorithm will be to maximize the throughput. The ultimate goal is to achieve a *constant throughput*, *i.e.,* to show that for any time round t, the number of rounds with at least a station still active is at most a constant factor higher than the number of stations activated until time t.

Finally, concerning the *energy consumption*, we evaluate the algorithm's efficiency in terms of the maximum, over all activated stations, number of transmissions performed by a single station.

1.1 Previous work and our contribution

Contention resolution on a shared channel is a classical problem in distributed computing that is getting a lot of attention recently. The first theoretical papers date back to the 70's and considered mainly deterministic solutions for the static scenario. Below we summarize the results most relevant to ours.

Deterministic algorithms. Capetanakis [6], Hayes [22], and Tsybakov and Mikhailov [40] independently presented a deterministic tree algorithm for conflict resolution in the model with collision detection accomplishing the task in $O(k + k\log(n/k))$ rounds, for every k and n. Surprisingly, if k (or a linear upper bound on it) is given *a priori* to the stations, then the same $O(k + k\log(n/k))$ bound can be achieved even non-adaptively, without collision detection, in simple channels with acknowledgments [25]. The proof is non-constructive; later Kowalski [26] showed a more constructive solution, based on selectors (cf., [14, 23]), reaching the same asymptotic bound. Clementi, Monti, and Silvestri [16] showed a matching lower

bound of $\Omega(k \log(n/k))$, which also holds for adaptive algorithms. If collision detection is available, there is an almost matching $\Omega(k \log n / \log k)$ lower bound (also valid for adaptive algorithms) demonstrated by Greenberg and Winograd [21]. All of the above results hold for the static scenario. In the dynamic scenario, De Marco and Kowalski [17] showed that $O(k \log n \log \log n)$ time rounds are sufficient to each station to transmit successfully with a nonadaptive deterministic algorithm. Interestingly, this almost matches the $\Omega(k \log n / \log k)$ lower bound in [21], although the latter holds in a much stronger setting: for adaptive algorithms, in the static scenario and with collision detection.

Randomized algorithms. As for randomized solutions, Greenberg, Flajolet and Ladner [19] and Greenberg and Ladner [20] presented an algorithm with collision detection working in $2.14k + O(\log k)$ rounds with high probability without any *a priori* knowledge of the number k of contenders. More recently, Fernández Anta, Mosteiro and Ramon Muñoz [1] obtained the same asymptotic (optimal) bound in the model without collision detection with a non-adaptive algorithm that also ignores any knowledge about contention size k. This shows that in the static model, *i.e.,* when all the packets arrive at the same time, there is no asymptotic difference in the time complexity between adaptiveness and non-adaptiveness, even in the absence of any knowledge about channel contention.

In the dynamic scenario considered in this paper, Bender et al. [2] designed an adaptive algorithm *with collision detection* that, without any given bound on parameter k, exhibits constant throughput, linear latency and $O(\log \log^* k)$ expected transmissions per station. Later, De Marco and Stachowiak [28] proved that constant throughput and linear latency can also be achieved, with high probability, even in the more severe setting *without collision detection*, although at the expenses of a higher energy cost. In a recent breakthrough, Bender et al. [3] improved the energy cost to $O(\log^2 k)$, while preserving both constant throughput and linear latency. They considered a more restricted setting where each station is obliged to leave the system once it broadcasts its message.

Related work. The contention resolution problem has been also studied in the more general framework of multi-hop radio networks, particularly in the context of problems such as (multi-)broadcast, gossip and others in the so-called blindfold model, *i.e.* in total absence of knowledge about topology and network parameters [9, 14, 29].

Developments where similar issues on selecting stations (included broadcasting in multi-hop radio networks) under many assumptions, mainly regarding knowledge and synchrony, can be found in [13, 12, 8, 10, 15, 17, 34, 33, 38, 37, 31, 32, 30].

Our contribution. In Section 2, we design a randomized adaptive algorithm which exhibits constant throughput and only a logarithmic (in the number of participating stations k, initially unknown to participants) energy cost. Our result holds with high probability in the number of participants and this number is unknown and potentially unbounded. The analysis is made against an adaptive adversary, c.f., Theorem 7. Exploiting the adaptivity of the algorithm, we allow the stations to stay in the system even after their successful transmission. Hence, our result improves on the polylogarithmic energy cost showed in [3] if stations are not obliged to switch off once they successfully transmit their packet, but can stay in the system communicating coordination information to the other stations.

Note that even if stations are allowed to stay in the system after their successful transmissions, they contribute to the throughput and energy cost; thus, in order to optimize these measures, we have to limit such stay and additional communication to absolute minimum, which is asymptotically negligible.

In Section 3 we show that any *non-adaptive* algorithm in the model with anonymous stations cannot achieve simultaneously throughput $\Omega\left(\frac{1}{\text{polylog}(k)}\right)$ whp and energy $O\left(\frac{\log^2 k}{(\log\log k)^2}\right)$ even with a constant probability, c.f., Theorem 8. Thus, they need an energy cost that is worse than that of our algorithm by an $\Omega\left(\frac{\log k}{(\log\log k)^2}\right)$ factor for every algorithm with throughput $\Omega\left(\frac{1}{\text{polylog}(k)}\right)$. Non-adaptive protocols are such that each station chooses a distribution of transmission rounds by itself without taking into account feedback from the channel. Such protocols do not assume randomly independent choices in rounds, therefore they include a wide class of algorithms such as backoff.

Our algorithmic approach. When designing a contention resolution algorithm for the *dynamic scenario*, one has to deal with the problem caused by new arrivals of stations that, being out of sync with stations activated earlier, can interfere with their transmissions producing collisions. This interaction between new and old stations plays a major role in any contention resolution algorithm and represents the most challenging obstacle. This is made even more difficult if one has to save the number of transmissions per station, so to keep the energy cost low.

In [3] this issue is overcome in an elegant way by allowing the older players to jam the new players, so avoiding interference from the newcomers. This is done by a clever interaction between the probabilities of real transmissions and the probabilities of jamming.

Our algorithm avoids the interference between old and new stations by keeping them in two separated groups that are coordinated by means of a leader that periodically sends information about the status of the system. There are many challenges that have been tackled by our solution.

One is to assure that a leader sends information periodically, while keeping the energy cost under a logarithmic threshold.

Another one comes from the fact that the adversary can partition the execution of the algorithm in several disjoint *activity intervals*, each of them characterized by its own set of contending stations. Estimating the *total* throughput of the whole execution required the development of a new technical tool (see the notion of *random variable condensed into a vector* in Section 2.2.2) for extending the analysis of throughput for a single activity interval (with results holding with high probability with respect to the contention of the single interval, *i.e.* the number of stations involved only in that interval) to the throughput for the union of all disjoint activity intervals (with results holding with high probability with respect to the total contention, *i.e.* the total number of participating stations during the whole execution).

Additionally, our approach involves several techniques for leader election, size approximation of the participating stations, testing efficiently whether a contention resolution has been accomplished. This also highlights interesting relationships between contention resolution and other classical problems in distributed computing.

Our lower bound approach. We construct and analyze different random wake-up patterns to prove that any correct non-adaptive contention resolution algorithm that wants to be efficient in terms of throughput has an energy cost $\Omega(\log^2 k/(\log\log k)^2)$.

We start from a first weaker lower bound $\Omega(\log k/\log\log k)$, which is guaranteed even for simple wake-up times uniformly distributed, and then we square this bound by building some more complex random wake-up instances.

More precisely, we first show that if the sum of transmission probabilities is above a logarithmic threshold in all rounds of an interval, then there are small chances of having a successful transmission in that interval. Then we show how to define random wake-up

patterns such that if the algorithm wants to keep a throughput $\Omega(\frac{1}{\log^a k})$, then it has to transmit as much as to keep the sum of transmission probabilities above the logarithmic threshold in the first $\frac{k}{\text{polylog}(k)}$ rounds. Therefore for some wake-up instances, $k - \frac{k}{\text{polylog}(k)}$ nodes already incur an energy cost $\Omega(\frac{\log k}{\log\log k})$.

Then, we could recursively "pump-up" the energy cost by recursively repeating the construction for the remaining $\frac{k}{\text{polylog}(k)}$ nodes.

1.2 Conventions and notation

By convention, we assume that a station is activated at round 0 of its local clock and can start transmitting from its local round number 1. At each round a station can decide the probability of transmission by means of a randomized algorithm. Since we are dealing with adaptive algorithms, these probabilities may depend on the history of the channel feedback and do not have to be independent over rounds.

Although there is no global time accessible to the stations, in the analysis we will need a *reference clock* (not visible to the stations) that allows us to argue about the behaviour of all the stations involved in the computation at a given moment.

For any time t of a given reference clock, we denote by $\hat{A}[t]$ the set of stations activated until time t. The transmission probability assigned by the protocol to a station $v \in \hat{A}[t]$ at time t will be denoted by $q_v[t]$. Some already activated stations, however, may not be active during the protocol execution in time t, because of switching off earlier; therefore, we use $A[t] \subseteq \hat{A}[t]$ to denote the set of stations that are *still active* at time t.

We define the *sum of transmission probabilities* at time t as follows: $\sigma[t] = \sum_{u\in A[t]} q_u[t]$.

Analogously, we will also need to consider the above sum over all activated stations until time t (*i.e.* not considering switches-off): $\hat{\sigma}[t] = \sum_{u\in \hat{A}[t]} q_u[t]$.
Surely, if t' is the time of the first successful transmission, then $\hat{\sigma}[t] = \sigma[t]$ for every $t < t'$.

In our analysis we will also need to deal with the sum of probabilities used by a station up to some time of its local clock. We define the *sum of transmission probabilities of an arbitrary station* up to local time i as: $s(i) = \sum_{j=1}^{i} p(j)$, where $p(j)$ denotes the transmission probability of the station at round j of its *local clock*. We do not need to specify the station which this probabilities refers to, as it will be clear from the context.

2 An adaptive algorithm for unknown contention

We now describe our protocol `AdaptiveCTLE`, which resolves the contention with constant throughput and logarithmic energy, without any knowledge on the number of contenders. The number of contenders could be arbitrary and they could be activated during an arbitrarily long time interval. Besides the data packet itself, each station can send a one-bit control message. For the sake of presentation we will refer to these control messages as <`D mode`> (encoded with bit 0) and <`any D-station left?`> (encoded with bit 1).

High-level description. The reader can refer to Figure 1 for a graphical representation of algorithm's behaviour. The algorithm alternates between two modes: a *leader election mode* (*L mode*) and a *dissemination mode* (*D mode*). The first mode aims at getting a synchronized subset of stations and electing a leader, which has the task of coordinating the computation in the next dissemination mode (it will also help in Protocol `Estimate&Increase`, which is a sub-routine executed in D mode). This is actually the contention resolution among the synchronized subset of stations defined in the preceding execution of the leader election

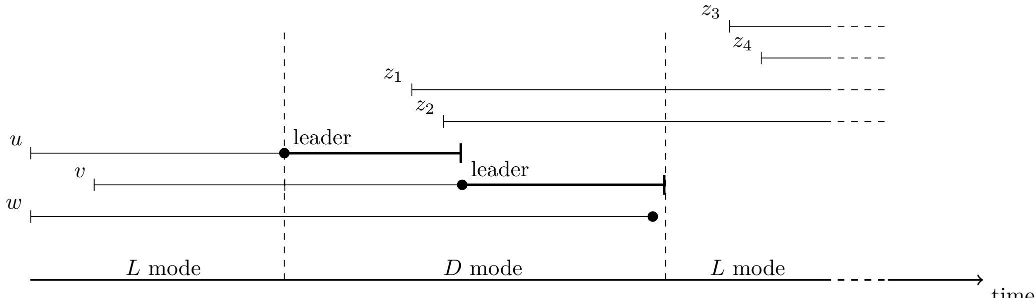


Figure 1 Horizontal segments represent the activity periods of stations, black circles indicate *first* successful transmissions for each station. Stations u, v and w start in L mode. Then, u is elected a leader: the stations are now synchronized and the D mode starts. Thicker segments show the change of leadership among stations every $O(\log k)$ rounds. Stations z_1 and z_2 wake up while u, v and w are in D mode: they remain *pending* until the current D mode is active (see the middle interval on the picture). Once u, v and w have switched off, z_1 and z_2 start a new L mode. Other stations (z_3 and z_4) can possibly join in arbitrary times. Once a leader has been elected a new D-mode starts and so forth. This process is iterated until a D mode ends and no station wakes up during its execution.

mode. The leader does not remain the same for the whole execution of the algorithm, but the stations take turns in this role, in order to keep the maximum number of transmissions below a logarithmic threshold (this will be assured by Protocol `SlowIncrease`).

All the stations running the D mode use a synchronized clock ticking rounds modulo 4. Odd rounds are used to execute the actual contention resolution protocol, while even rounds will be used together with the leader to learn and send additional information to the participating stations.

At any time, the system is either in L mode or in D mode. A station involved in the L mode (resp., D mode) will be called an L-station (resp., a D-station). A newly awaken station learns the mode of the system during the first 4 rounds. Then it remains with an empty status until it keeps receiving, once every 4 rounds, a message <`D mode`> from the current leader. This message informs the newcomers that the system is busy with the dissemination mode. We call these stations with an empty status *pending stations*. This status of “pending” will last as long as the stations that are currently in D mode have not switched-off. It is the leader that will inform when this happens by ceasing to send the <`D mode`> message and switching off. When this happens, all the pending stations start a new L mode, and so forth. This process is iterated until no new station is injected in the system; more precisely, when, after the switching-off of all the current D-stations, no pending station is waiting to start a new L mode.

A formal description of the algorithm can be found in the following pseudocodes.

2.1 Pseudocodes

The main protocols of the algorithm are `AdaptiveCTLE` (Protocol 1), which is the first protocol executed by a station when it is activated, and `AdaptiveLeader` (Protocol 2), which is executed by the leader.

Protocol 1 and 2 (`AdaptiveCTLE` and `AdaptiveLeader`). A newly awaken station u starts with the execution of Protocol `AdaptiveCTLE`. It takes the status of pending station and keeps it while in the loop in line 2 of this protocol, *i.e.,* while it keeps receiving the <`D`

Algorithm 1 `AdaptiveCTLE` (*executed by any station u*).

STATUS $\leftarrow \emptyset$
while STATUS $\notin \{L, D\}$ **do** // *a woken up station keeps waiting (pending station)*
  listen to the channel
  if u does not receive message <`D mode`> in 4 consecutive rounds **then**
    STATUS $\leftarrow L$ // *u becomes an L-station*
while u is active **do**
  if STATUS $= L$ **then**
    execute `DecreaseSlowly` // *the first successful station becomes the leader*
    if u has been elected the leader **then**
      execute `AdaptiveLeader` // *see Protocol 2*
    STATUS $\leftarrow D$ // *once a leader is elected, station switches to dissemination mode*
    `time_counter` $\leftarrow 0$ /* *now all awaken stations are synchronized and `time_counter` will denote the current round number started at the time the leader has been elected* */
  if STATUS $= D$ **then**
    if `time_counter` is odd **then**
      execute round $\lfloor($`time_counter`$+1)/2\rfloor$ of `Estimate&Increase`(c) (switch-off at the first successful transmission)
    else if `time_counter` $= 2^x$, for some integer $x \geq 2$, **then**
      transmit message <`any D-station left?`>

Algorithm 2 `AdaptiveLeader` (*executed by the leader*).

1: **while** u is the leader **do**
2:   **if** `time_counter` is odd **then**
3:     participate as leader in `Estimate&Increase`(c)
4:   **else if** `time_counter` $= 2^x$, for some $x \geq 2$, **then**
5:     transmit message <`any D-station left?`>
6:     **if** the message is received **then** // *if D mode has terminated*
7:       switch off
8:   **else** transmit <`D mode`> // *leader without acknowledgment => D mode continues*

Algorithm 3 `DecreaseSlowly` (*executed by a station u*) [24].

1: $q \leftarrow$ some constant > 0
2: $i \leftarrow 0$
3: **while** u is active **do**
4:   transmit the message with probability $q \cdot \frac{1}{2q+i}$
5:   **if** transmission is successful **then**
6:     become a leader
7:   $i \leftarrow i + 1$

Algorithm 4 `Estimate&Increase`(c) (*executed by a station u*).

1: **while** `Test` is true **do**
2:   Execute `Estimate` to obtain an estimate value k
3:   Execute `SlowIncrease`(k, c)

`mode`> message once every 4 rounds. If within 4 consecutive rounds of waiting, station u does not get the <`D mode`> message, then the system is not in D mode and three cases may have occurred:

(a) there was no active station when u has been woken up;
(b) the system was in L mode when u has been woken up;
(c) a D mode was running when u has been woken up, but all D-stations delivered their messages and switched off.

In all three cases, the station exits the loop in line 2 as an L-station and starts the subsequent loop. Being an L-station, it starts executing protocol `DecreaseSlowly` (cf. the pseudocode of Protocol 3). This is a wake-up protocol introduced in [24] whose goal is to get just one successful transmission among an asynchronized set of stations.

Once a successful transmission appears in some round t, the station which transmitted in t becomes the *leader*. At this point the leader and all other stations that were alive at round t, are synchronized. They set up a new variable `time_counter` to 0 and a D mode starts: the leader starts the execution of Protocol `AdaptiveLeader`, while the other synchronized stations continue with the execution of `AdaptiveCTLE`.

Let us denote by C such a *synchronized* subset of stations (not including the leader). We can assume that a global clock (represented by variable `time_counter` initiated by the leader) starts for all the stations in C at the round in which the leader was elected. This allows us to use a contention resolution protocol for a synchronized set of stations. We accomplish this task with Protocol 4, `Estimate&Increase`, which guarantees that all synchronized stations in C transmit successfully within $O(|C|)$ rounds after the synchronization round whp in $|C|$. During an execution of `Estimate&Increase`, all stations from set C switch off directly after a successful transmission. In order for such a protocol to work properly, it is necessary to avoid that stations that have woken up during its execution (newcomers) could interfere disturbing the transmissions.

In order to make it possible that the newcomers understand what is happening in the system, the algorithm `Estimate&Increase` is executed in odd rounds only, while even rounds are devoted to performing the following kind of coordination. In rounds $t = 2^x$, for some integer $x \geq 2$, the D-stations and the leader send message <`any D-station left?`>, as stated in lines 17 and 5 of their respective Protocols 1 and 2 (note that this requires a number of transmissions that is only logarithmic in the length of protocol `Estimate&Increase`, that is $O(\log |C|)$).

This message will be successfully heard by the leader (and delivered to the pending stations busy in the loop of line 2) if and only if the leader is the only transmitter, that is, when all the D-stations in set C have switched off. If the transmission is successful, the leader switches off immediately. If, on the other hand, this message is not successfully acknowledged, then it means that there is still some D-station active in the system. In this case, in all subsequent even rounds that are not reserved to message <`any D-station left?`>, the leader sends message <`D mode`> informing the pending stations that the D mode has not terminated, and so they have to keep waiting (*i.e.,* listening) in the loop.

We can now see how the above strategy guarantees that any station activated during the execution of protocol `Estimate&Increase`, remains pending in the loop of line 2 silently waiting for its termination and then it exits the loop as an L-station. Indeed, while `Estimate&Increase` is running, the leader transmits a <`D mode`> message once every at most 4 rounds (all even rounds with the exception of rounds 2^x, for $x \geq 2$). This keeps the station waiting in the loop. Once, the D-mode has terminated, the pending station hears no <`D mode`> message during 4 consecutive rounds and therefore it exits the loop as an L-station, starting a new L-mode. This process is iterated until it happens that no pending station is waiting in the loop of line 2, *i.e.* when no other station is injected in the system.

Protocol 3 (`DecreaseSlowly`). This protocol is used to elect a leader among the set of *asynchronized* stations running in L mode. For this task we use a wakeup protocol introduced in [24]. This protocol let just one station to successfully transmit. This station takes the role of a leader.

Protocol 4 (`Estimate&Increase`). The purpose of this protocol is to solve the contention among the *synchronized* set of stations running in D mode. The leader will also participate to the computation (as stated on line 3 of Protocol `AdaptiveLeader`). The purpose is accomplished by means of a sequence of two protocols: first, `Estimate`, which computes a 2-approximation of the number of synchronized contenders, and subsequently, `SlowIncrease`, which uses the previously computed estimate to let the stations to successfully transmit their messages and switch off. The two protocols `Estimate` and `SlowIncrease` will be executed repeatedly until all the stations have switched off. This condition is checked by a function `Test`. Let us now describe in more details the behavior of each of these three ingredients.

Protocol `Estimate`. In order to compute our 2-approximation of the number of contenders, we adapt Algorithm $(1+\epsilon)$`-approximation` for the beeping model [5] to our shared channel without collision detection. Because in our model beeps are not explicitly recognized by the channel, we emulate each round r of Algorithm $(1+\epsilon)$`-approximation` by the following two `echo` rounds (see [27]) in our model:

- round $2r$ of Protocol `Estimate`: if a participating station is scheduled to transmit in the corresponding round r of $(1+\epsilon)$`-approximation`, it also transmits in round $2r$ of `Estimate`;
- round $2r+1$ of Protocol `Estimate`: if a participating station is scheduled to transmit in the corresponding round r of $(1+\epsilon)$`-approximation`, it also transmits in round $2r+1$ and the leader transmits as well.

If nothing is heard in the first `echo` round and the leader is heard in the second `echo` round, then it means there is no beep in round r of $(1+\epsilon)$`-approximation`; otherwise there is a beep.

Protocol `SlowIncrease`(k, c). Once we have obtained an approximate estimate of the number of participating stations, we can use an algorithm for contention resolution which exploits such an information. For this task we can use protocol `NonAdaptiveWithK` (k, c) [28], which uses an upper bound of the number k of contenders and a sufficiently large constant c (which determines the probability of success). Each successfully transmitting station automatically swaps the leadership role with the current leader, which switches off.

In order to assure logarithmic energy of the leader, if there is no swap in consecutive $\log k$ rounds of the original protocol `NonAdaptiveWithK` (k, c) [28], the leader runs a tournament that elects another leader in $O(\log k')$ rounds, where k' is the actual number of participants. The binary search with `echo` protocol from [27] is used, coordinated by the leader, in the beginning of which stations choose random ids from interval $[1, O(k)]$. It is possible that more than one station selects the same id, in which case the leader discovers an echo for that value, and runs the tournament again but this time only for the colliding stations. If the time of the tournament exceeds $2\log k$, its actual length divided by 2 is set as the power of 2 in the new estimate of k – the number of participating stations. Note that the time, and thus also the energy, of the tournament is amortized by the value of $\log k$ of the estimate, and so the following number of rounds in which the original protocol `NonAdaptiveWithK` (k, c) is executed.

Function `Test`. Similarly as in the protocol `Estimate`, the same `echo` procedure (a single 2-round execution of it) is used with respect to the remaining participants of the superior protocol and the current leader. If there is a beep, the test is true, otherwise it returns false.

2.2 Analysis of throughput and energy

Correctness and time complexity of our algorithm `AdaptiveCTLE` will be finally proved in Theorem 7. We will go through the following steps.

First, we analyze algorithm `AdaptiveCTLE` in an *activity interval*, *i.e.* a maximal size interval of consecutive active rounds. In doing so, we start from the analysis of the basic sub-routine protocols (Section 2.2.1), and then put them together into the analysis of the whole activity interval (Section 2.2.2).

Next, we put the disjoint activity intervals together, using their independence and partitioning the activity of the adversary into classes of sub-adversaries, one for each activity interval (Section 2.2.3).

2.2.1 Analysis of protocols

Consider an activity interval with k contenders. The first step is to analyze the performance of procedure `DecreaseSlowly`. In [24], an algorithm *Decrease Slowly* has been presented for the first time, and it was proven to solve the wake-up problem (*i.e.*, allow just one successful transmission) in $O(k \log k)$ rounds whp(k). In [28] it was improved to $O(k)$ rounds whp(k). Both analysis assumed non-adaptive adversary, *i.e.*, an adversary scheduling the wake-up times in advance. In the following lemma, we show more detailed properties of this algorithm, under a stronger adaptive adversary (as considered in this paper). The proof is deferred to the appendix.

▶ **Lemma 1.** *Algorithm* `DecreaseSlowly` *finishes wake-up in* $O(k)$ *rounds and with* $O(\log k)$ *energy whp*(k)*, where* k *is the number of activated stations during the activity interval of this execution. Moreover, for any* $k' > k$*, algorithm* `DecreaseSlowly` *finishes wake-up in* $O(k \log(k'/k))$ *rounds and with* $O(\log k')$ *energy whp*(k')*.*

The two ingredient protocols of algorithm `Estimate&Increase` satisfy the following:

▶ **Lemma 2** ([5])**.** *Protocol* `Estimate` *outputs a* 2*-approximation of the number* k *of stations in D-mode in* $O(\log k)$ *rounds and* $O(\log k)$ *energy whp(*k*), and for any* $k' > k$*, it outputs a* 2*-approximation of the number* k *of stations in D-mode in* $O(\log k')$ *rounds and energy whp(*k'*).*

▶ **Lemma 3** ([28])**.** *Protocol* `SlowIncrease`(k, c) *solves the contention resolution problem for at most* k *stations in D-mode in* $O(k)$ *rounds and with* $O(\log k)$ *energy whp(*k*).*

▶ **Lemma 4.** *Algorithm* `Estimate&Increase`(c) *solves the contention resolution problem within an activity interval in* $O(k)$ *rounds and with* $O(\log k)$ *energy whp*(k)*, where* k *is the number of D-stations in the beginning of this execution. Moreover, for any* $k' > k$*, algorithm* `Estimate&Increase`(c) *solves the contention resolution problem within an activity interval in* $O(k \log(k'/k))$ *rounds and with* $O(\log k')$ *energy whp*(k')*.*

Proof. The first part follows from putting together Lemmas 2 and 3. The second part follows from repeating the above independently $O(\log(k'/k))$ times, until the `Test` becomes false and the algorithm stops. The standard probabilistic argument applies here because of two reasons: these repeating parts are synchronized by the `Test` run in the beginning of the loop, and stations participate in only a single iteration of the loop.

In the Protocol `Estimate&Increase` and the associated confirmation rounds, there is a constant number of transmissions per one successful transmission, on average, thus $O(\log k)$ whp(k) and $O(\log k')$ whp(k') in an execution of length $O(k \log(k'/k))$. ◀

2.2.2 Analysis of a single activity interval

In order to show that the performance guarantees of our algorithm hold with high probability with respect to the total number of stations awaken during any execution, independently of how the execution is partitioned into disjoint activity intervals, we introduce the notion of condensed random variables. Next, in Lemma 5, whose proof is in the appendix, we use such a tool to show that the sum of the lengths of the activity intervals is bounded with high probability with respect to the total number of stations awaken.

Let $c > 0$ be a sufficiently large constant, depending on the exponent η in the formula for whp(). We say that positive integer random variables ℓ_i, for $1 \leq i \leq x$, are *condensed* into a vector $\kappa = (w_1, \ldots, w_y)$, for some positive integer y such that $w_1 \leq \ldots \leq w_y = x$, if for every $1 \leq j \leq y$, set $L_j = \{i : E[\ell_i] = O(2^j)$ & $\ell_i \leq c \cdot 2^j$ holds whp(2^j) & $\ell_i \leq c \cdot 2^j \log(\bar{\kappa}/2^j)$ holds whp$(\bar{\kappa})\}$ is of size w_j, where $\bar{\kappa} = c \cdot \sum_{j=1}^{y} 2^j w_j$.

The following technical fact holds (see the appendix for the proof).

▶ **Lemma 5.** *Let ℓ_i, for $1 \leq i \leq x$, be random variables condensed into $\kappa = (w_1, \ldots, w_y)$, for some positive integer y. Then, $\sum_{i=1}^{x} \ell_i = O(\bar{\kappa})$ whp$(\bar{\kappa})$.*

We now show the performance guarantees of our algorithm within each activity interval.

▶ **Lemma 6.** *Algorithm* `AdaptiveCTLE` *solves the contention resolution problem within an activity interval in $O(k)$ rounds and with $O(\log k)$ energy whp(k), where k is the total number of stations activated during this execution. Moreover, for any $k' > k$, algorithm* `AdaptiveCTLE` *solves the contention resolution problem within an activity interval in $O(k \log k')$ rounds and with $O(\log k')$ energy whp(k').*

Proof. Let I be the interval of rounds involved in this execution of `AdaptiveCTLE`. The system starts in L mode (apart from an initial waiting period of at most 4 rounds spent in the first `while` loop of Protocol `AdaptiveCTLE`, during which the newly activated stations, as pending stations, realize that the system was inactive) and then it enters the D mode.

The execution of the algorithm is composed of a sequence of L mode/D mode executions, in such a way that the ith L mode is followed by the ith D mode. Each D mode execution involves the same set of stations that participated to the previous L mode, and the ith L mode execution, for $i > 1$, involves the stations that were pending in the previous D mode. This will continue until the end of interval I which occurs when all the D-stations of the last execution of `Estimate&Increase` switch off and there is no station pending at the end of it.

Let x be the number of such L mode/D mode executions. For $1 \leq i \leq x$, let ℓ_i^L be the length of the ith L mode and ℓ_i^D be the length of the following D mode. By Lemma 1 we have that $\ell_i^L = O(m_i)$ whp(m_i), where m_i is the number of stations that participated in the ith L mode. This is also the number of stations involved in the next D mode. By Lemma 4, it follows that (1) $\ell_i^D = O(m_i)$ whp(m_i), (2) $E[\ell_i^D] = O(m_i)$ and (3) $\ell_i^D = O(m_i \log(k/p_i))$. Hence, considering the lengths $\ell_i^{LD} = \ell_i^L + \ell_i^D$ of each L mode/D mode execution, we also have (1) $\ell_i^{LD} = O(m_i)$ whp(m_i), (2) $E[\ell_i^{LD}] = O(m_i)$ and (3) $\ell_i^{LD} = O(m_i \log(k/p_i))$.

Let w_j be the number of executions i such that $m_i \leq c \cdot 2^j$. It follows that the random variables ℓ_i^{LD}, for $1 \leq i \leq x$, are condensed into $(w_1, w_2, \ldots, w_y)$. By Lemma 5 we have that $\ell = \sum_{j=1}^{x} \ell_j^{LD} = O(\sum_{j=1}^{x} m_j) = O(k)$ whp(k).

Lemmas 1 and 4 guarantee $O(\log k)$ energy whp(k) for the stations except the activity of the leader in Protocol 1. It is easy to see that the transmissions of the leader could continue for a large period of time, which could go over the desired $O(\log k)$ upper bound. This is due to the total execution time of Protocol `Estimate&Increase` called on line 15, during which the leader keeps sending messages (along with the other synchronized stations) until it gets an acknowledgement. However, the algorithm resolves this issue by requesting the stations participating to `Estimate&Increase` swapping the role of leader in such a way that none of them transmits for more than $O(\log k)$ rounds, see description of Protocol `SlowIncrease`(k, c). The second part of the lemma follows directly by applying second parts of the results for the ingredient protocols (for any $k' > k$), in Lemmas 1 and 4, and estimating each $\log(k'/p_i)$ from above by $\log k'$. ◀

2.2.3 Putting activity intervals together

Finally, we are ready for the main theorem of this section. We will be using Lemma 5 again, where the ℓ_i's correspond now to the lengths of subsequent activity intervals and $\bar{\kappa}$ is an upper linear estimate of the activated stations in the whole considered execution.

▶ **Theorem 7.** *Algorithm* `AdaptiveCTLE` *has constant throughput and* $O(\log k')$ *energy, whp*(k')*, where* k' *is the total number of awaken station in a considered prefix of the execution of Algorithm* `AdaptiveCTLE`*.*

Proof. Fix an arbitrary execution of the algorithm. For any time round t, let $Q[t]$ be the set of rounds $r \leq t$ at which there is at least a station still active and $n[t]$ be the total number of stations activated until time t. We need to show that the size of $Q[t]$ is at most a constant factor higher than $n[t]$.

Depending on the distance between consecutive activation times (that are controlled by the adversary and can be arbitrarily large) the execution of the algorithm can be split into several independent executions involving disjoint subsets of stations. At the end of each execution, all the stations awaken during it will be switched-off. The time rounds between two consecutive executions do not belong to $Q[t]$.

Therefore, there exists an integer $1 \leq x \leq k$ such that $Q[t]$ can be partitioned into disjoint time intervals $I_1, \ldots, I_x$, each of them corresponding to an independent execution of the algorithm on a set S_i of stations, where all these subsets form a partition of the set of all the stations activated until time t, formally $\bigcup_{1 \leq i \leq x} S_i = \hat{A}[t]$ and $S_i \cap S_j = \emptyset$ for $i \neq j$. We apply Lemma 5 to these intervals in exactly the same way as we applied it to the independent executions of L mode and `Estimate&Increase` inside activity interval in the first part of the proof of Lemma 6. Now, the base properties of the lengths of activity intervals are guaranteed by Lemma 6, where k' stands for the total number of awaken stations in the considered prefix of execution, and Lemma 5 implies the theorem for k' being upper bounded by $\bar{\kappa}$ with respect to a constant factor.

Finally, note that due to the independence of the activity intervals, each awaken station participates in only one of them, therefore each station's energy is $O(\log k')$ whp(k'), by Lemma 6 and the union bound over the participating k' stations. ◀

3 A trade-off between throughput and energy of non-adaptive algorithms

An arbitrary sequence of k activation times for k stations will be called an *instance of at most k stations* and denoted by $I(k)$. For any instance $I(k)$, we will use a reference clock starting when the first station is activated. All the following rounds t refer to this clock.

Given an algorithm $\mathcal{A}$, we let $T_{\mathcal{A}}(I(k))$ be the maximum time needed for $\mathcal{A}$ to assure a successful transmission of any station in instance $I(k)$ whp. Analogously, we let $E_{\mathcal{A}}(I(k))$ be the expected number of transmissions per station spent by algorithm $\mathcal{A}$ on instance $I(k)$.

▶ **Theorem 8.** *Let $\tau(x) = \Omega(1/(\log^a x))$ for any constant $a > 0$. There exists an instance $I(k)$ such that any non-adaptive algorithm not knowing k and achieving throughput $\tau(k)$ whp, requires $\Omega(\log^2 k/(\log\log k)^2)$ expected transmissions per station.*

Proof of Theorem 8. In order to prove the theorem, from now on we fix an arbitrary non-adaptive algorithm $\mathcal{A}$ achieving throughput $\tau(k) = \Omega(1/(\log^a x))$ whp. All the following results are meant to hold for such an algorithm $\mathcal{A}$. Also, to simplify the description, all bounds involving throughput are meant to hold with high probability even when not explicitly stated. Moreover, we denote by $T(k)$ (resp. $E(k)$) the maximum $T_{\mathcal{A}}(I(k))$ (resp. $E_{\mathcal{A}}(I(k))$) taken over all instances $I(k)$ activating k stations.

▶ **Fact 9.** $T(k) \leq k/\tau(k)$.

Proof. Suppose on the contrary that $T(k) > k/\tau(k)$. Then for $t = T(k)$, the ratio between the number of activated stations and the number of active rounds will be

$$\frac{n[t]}{r[t]} \leq \frac{k}{T(k)} < \frac{k \cdot \tau(k)}{k} = \tau(k),$$

which contradicts the assumption on the throughput of algorithm $\mathcal{A}$. ◀

We start with the following lemma showing a first lower bound on the average number of transmissions. We will improve such a bound later on.

▶ **Lemma 10.** $E(k) = \Omega(\log k/\log\log k)$.

Proof. Let us build a random instance of k stations as follows. By the hypothesis on throughput and Fact 9, we can assume that $T(k) \leq k/\tau(k)$ whp. Let v be a station activated at time 1 of the instance. By definition of $T(k)$, v has to transmits successfully within $T(k)$ rounds whp. The number of rounds at which v transmits in the time period $[1, T(k)]$ is a random variable X such that

$$E(X) = E(k) = s(T(k)) = \sum_{i \in [1, T(k)]} p(i) \ .$$

By Markov's inequality we have

$$\Pr\left(X < 2E(X)\right) > 1/2 \ . \tag{1}$$

We now let the activation times of the other $k-1$ stations be distributed uniformly at random among the $T(k)$ rounds. Each station transmits with probability $p(1)$ at the first round it switches on. Since the algorithm does not know k, the probability $p(1)$ does not depend on k. Therefore, we have that at any round of $[1, T(k)]$ in which v transmits, the probability that this transmission is not successful is the probability that any of the other $k-1$ stations transmits at the same time, that is $(k-1) \cdot p(1) \cdot (1/T(k)) = \Omega(\tau(k))$, where the asymptotic bound is due to the inequality $T(k) \leq k/\tau(k)$. Thus, the probability of no successful transmission for station v during the whole interval $[1, T(k)]$ is at least $\Omega\left(\tau(k)^X\right)$.

Hence, this probability is at most $1/k^\eta$ for any predetermined constant $\eta > 0$, only when the number of transmissions of v is $X = \Omega(\log k/\log(1/\tau(k))) = \Omega(\log k/\log\log k)$. By Equation (1), it follows that $E(X) = \Omega(\log k/\log\log k)$ and the lemma follows. ◀

Now we show that if the algorithm transmits as much to keep the sum of transmission probabilities above a logarithmic threshold, then the probability of having a successful transmission becomes very low. Notice that before the first successful transmission occurs, we have $A[t] = \hat{A}[t]$. For this reason, in the following calculations we consider bounds on $\hat{\sigma}[t]$ instead of $\sigma[t]$, as they are equivalent until the first successful transmission appears.

▶ **Lemma 11.** *Let $T \leq k^2$. There exists a constant $\gamma > 0$ such that if $\hat{\sigma}[t] \geq \gamma \log k$ for every $t \in [1, T]$, then the probability of having at least one successful transmission in the time interval $[1, T]$ is smaller than $1/k$.*

Proof. The probability of having *at least one* successful transmission in the time interval $[1, T]$, is equivalent to the probability of having the *first* transmission in any round t of this interval. For a fixed round t, this probability is at most

$$\sum_{v \in \hat{A}[t]} p(t - t_v) \prod_{w \in \hat{A}[t], w \neq v} (1 - p(t - t_w)) \leq \hat{\sigma}[t] e^{-\hat{\sigma}[t]+1} ,$$

which can be made smaller than $1/k^3$, for a sufficiently large γ. By taking the union bound over all the $T \leq k^2$ rounds of the interval, we get that this probability is at most $1/k$. ◀

The following lemma shows that the hypothesis on throughput implies that the sum of transmission probabilities will be maintained above the logarithmic threshold determined in the previous lemma. In other words, if the algorithm wants to be efficient in terms of throughput, then it has to lose in terms of energy.

▶ **Lemma 12.** *Let γ be the constant determined in Lemma 11. There exists a constant $c > 0$ and an instance $I_1(k)$ such that, for k sufficiently large, $\hat{\sigma}[t] \geq \gamma \log k$ in all rounds $t \in [1, T(k/(c \log^{1+a} k))]$.*

Proof. By the hypothesis on throughput and Fact 9 we know that $T(k) \leq O(k \log^a k)$. Hence, $T(k/\log^{1+a} k) \leq O(k/\log k)$. Consequently, for any constant $c' > 0$ there exists a constant $c > 0$ such that $T(k/(c \log^{1+a} k)) \leq k/(c' \log k)$, for k sufficiently large.

Construction of instance $I_1(k)$. Letting $c' = \gamma/p(1)$, we can construct an instance $I_1(k)$ for k contending stations as follows. In each round $t \in [1, T(k/(c \log^{1+a} k))]$ we switch on $c' \log k$ stations. Note that for this task, it is sufficient to activate at most k stations, indeed:

$$c' \log k \cdot T\left(\frac{k}{c \log^{1+a} k}\right) \leq c' \log k \cdot \frac{k}{c' \log k} = k .$$

Each station transmits with probability $p(1)$ in the round it is switched on. Therefore, in every $t \in [1, T(k/(c \log^{1+a} k))]$, $\hat{\sigma}[t] \geq c' \log k \cdot p(1) = \gamma \log k$. ◀

Now we can show that we can build an instance of k stations such that the transmissions are mainly distributed at the end of the considered interval of $T(k)$ rounds.

▶ **Lemma 13.** *For some constant c, $E(k) - E\left(\frac{k}{c \log^{1+a} k}\right) = \Omega\left(\frac{\log k}{\log \log k}\right)$.*

Proof. In order to prove the lemma, we build a corresponding instance $I_2(k)$ and analyze its properties. We start as follows. Let γ be the constant determined by Lemma 11 and take an instance $I_1(k/2)$ of $k/2$ stations as guaranteed by Lemma 12. This lemma implies that $\hat{\sigma}[t] \geq \gamma \log(k/2)$ in all rounds $t \in \left[1, T\left(\frac{(k/2)}{c' \log^{1+a}(k/2)}\right)\right]$, for some constant c'. There exists

a constant c such that $T\left(\frac{(k/2)}{c'\log^{1+a}(k/2)}\right) \geq T\left(\frac{k}{c\log^{1+a}k}\right)$. Therefore, by Lemma 11, there is no successful transmission in all rounds $1, 2, \ldots, T\left(\frac{k}{c\log^{1+a}k}\right)$ whp. Hence, we can conclude that whp a station v starting the protocol at round 1 is not able to transmit successfully in the interval $\left[1, T\left(\frac{k}{c\log^{1+a}k}\right)\right]$.

Now we continue the construction of an instance $I_2(k)$ by distributing uniformly at random the remaining $k/2$ stations in the interval $[T(k/(c\log^{1+a}k), T(k)]$. The non-adaptive algorithm $\mathcal{A}$ assigns a fixed sequence of transmissions to v in this interval. Recalling Fact 9, each of these transmissions is not successful with probability larger than $(k/2)\cdot p(1)\cdot(1/T(k)) = \Omega(\tau(k)) = \Omega(1/\log^a k)$.

Thus, in order to have a successful transmission with high probability, station v needs to transmit $\Omega(\log k/\log\log k)$ times in the interval $[T(k/(c\log^{1+a}k)), T(k)]$, as $(1/\log^a k)^{\Omega(\log k/\log\log k)} = 1/\text{poly}(k)$.

Thus, analogously as in the proof of Lemma 10, in order to have a successful transmission with high probability, station v needs to transmit $\Omega(\log k/\log\log k)$ times between round $T(k/(c\log^{1+a}k))$ and $T(k)$. Therefore, $E(k) - E(k/(c\log^{1+a}k)) = \Omega(\log k/\log\log k)$. ◀

Finally, the next lemma concludes the proof of Theorem 8.

▶ **Lemma 14.** $E(k) = \Omega(\log^2 k/(\log\log k)^2)$.

Proof. We can write down a telescoping sum, where c is the constant determined in Lemma 13:

$$\begin{aligned} E(k) &= \left(E(k) - E(k/(c\log^{1+a}k))\right) + \left(E(k/(c\log^{1+a}k)) - E(k/(c\log^{1+a}k)^2)\right) + \\ &\quad + \left(E(k/(c\log^{1+a}k)^2) - E(k/(c\log^{1+a}k)^3)\right) + \ldots \end{aligned}$$

The thesis follows by noting that this sum has $\Omega(\log k/\log\log k)$ terms, and the first half of these terms are $\Omega(\log k/\log\log k)$ by Lemma 13. ◀

4 Open problems

The most interesting open direction is to study tradeoff between energy consumption and other measures. In particular, is logarithmic energy necessary for anonymous shared channel against adaptive adversary in order to achieve constant throughput? If so, could we lower the energy requirement by allowing slightly smaller throughput, and if so, how much smaller? How the performance changes if we start restricting protocols, for instance, by limiting randomness or/and number of listening rounds, not allowing any control bits or requesting successful stations to disconnect immediately. Considering occasional failures and/or accounting rounds when stations actively listen to the energy measure are examples of other challenging directions.

References

1 A. Fernández Anta, M. A. Mosteiro, and J. Ramon Mu noz. Unbounded contention resolution in multiple-access channels. *Algorithmica*, 67:295–314, 2013.

2 M. A. Bender, T. Kopelowitz, S. Pettie, and M. Young. Contention resolution with log-logstar channel accesses. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing (STOC)*, pages 499–508, Cambridge, MA, USA, 2016. ACM.

3 Michael A. Bender, Tsvi Kopelowitz, William Kuszmaul, and Seth Pettie. Contention resolution without collision detection. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2020, pages 105–118, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3357713.3384305`.

4 Giuseppe Bianchi. Performance analysis of the ieee 802.11 distributed coordination function. *Selected Areas in Communications, IEEE Journal on*, 18:535–547, April 2000. doi:10.1109/49.840210.

5 Philipp Brandes, Marcin Kardas, Marek Klonowski, Dominik Pająk, and Roger Wattenhofer. Fast size approximation of a radio network in beeping model. *Theoretical Computer Science*, 810:15–25, 2020. Special issue on Structural Information and Communication Complexity. doi:10.1016/j.tcs.2017.05.022.

6 J. Capetanakis. Tree algorithms for packet broadcast channels. *IEEE Transactions on Information Theory*, 25:505–515, 1979.

7 B. S. Chlebus. Randomized communication in radio networks. In P. M. Pardalos, S. Rajasekaran, J. H. Reif, and J. D. P. Rolim, editors, *Handbook on Randomized Computing*, pages 401–456. Springer, New York, NY, USA, 2001.

8 Bogdan S. Chlebus, Leszek Gąsieniec, Dariusz R. Kowalski, and Tomasz Radzik. On the wake-up problem in radio networks. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Automata, Languages and Programming*, pages 347–359, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

9 Bogdan S. Chlebus, Leszek Gąsieniec, Alan Gibbons, Andrzej Pelc, and Wojciech Rytter. Deterministic broadcasting in unknown radio networks. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '00, pages 861–870, USA, 2000. Society for Industrial and Applied Mathematics.

10 Bogdan S. Chlebus and Dariusz R. Kowalski. A better wake-up in radio networks. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 266–274, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/1011767.1011806.

11 Bogdan S. Chlebus, Gianluca De Marco, and Dariusz R. Kowalski. Scalable wake-up of multi-channel single-hop radio networks. *CoRR*, abs/1411.4498, 2014. arXiv:1411.4498.

12 Bogdan S. Chlebus, Gianluca De Marco, and Dariusz R. Kowalski. Scalable wake-up of multi-channel single-hop radio networks. *Theor. Comput. Sci.*, 615:23–44, 2016. doi:10.1016/j.tcs.2015.11.046.

13 Bogdan S. Chlebus, Gianluca De Marco, and Muhammed Talo. Naming a channel with beeps. *Fundam. Informaticae*, 153(3):199–219, 2017. doi:10.3233/FI-2017-1537.

14 M. Chrobak, L. Gasieniec, and W. Rytter. Fast broadcasting and gossiping in radio networks. *Journal of Algorithms*, 43:177–189, 2002.

15 Marek Chrobak, Leszek Gasieniec, and Dariusz Kowalski. The wake-up problem in multi-hop radio networks. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, pages 992–1000, USA, 2004. Society for Industrial and Applied Mathematics.

16 A. E. F. Clementi, A. Monti, and R. Silvestri. Distributed broadcast in radio networks of unknown topology. *Theoretical Computer Science*, 302:337–364, 2003.

17 G. De Marco and D. Kowalski. Fast nonadaptive deterministic algorithm for conflict resolution in a dynamic multiple-access channel. *SIAM J. Comput*, 44(3):868–888, 2015.

18 Robert G. Gallager. A perspective on multiaccess channels. *IEEE Trans. Information Theory*, 31(2):124–142, 1985.

19 A. G. Greenberg, P. Flajolet, and R. E. Ladner. Estimating the multiplicities of conflicts to speed their resolution in multiple access channels. *Journal of the ACM*, 34(2):289–325, 1987.

20 A. G. Greenberg and R. E. Ladner. Estimating the multiplicities of conflicts in multiple access. In IEEE, editor, *Proc. of the 24th Annual Symp. on Foundations of Computer Science (FOCS) (Tucson, AZ.)*., pages 383–392, Tucson, AZ, USA, 1983. IEEE.

21 A. G. Greenberg and A S. Winograd. lower bound on the time needed in the worst case to resolve conflicts deterministically in multiple access channels. *Journal of ACM*, 32:589–596, 1985.

22 J. F. Hayes. An adaptive technique for local distribution. *IEEE Transactions on Communications*, 26:1178–1186, 1978.

23 P. Indyk. Explicit constructions of selectors and related combinatorial structures. In *Proceedings, 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 697–704, San Francisco, CA, USA, 2002. ACM-SIAM.

24 T. Jurdzinski and G. Stachowiak. Probabilistic algorithms for the wakeup problem in single-hop radio networks. *Theory Comput. Syst*, 38(3):347–367, 2005.

25 J. Komlós and A. G. Greenberg. An asymptotically optimal nonadaptive algorithm for conflict resolution in multiple-access channels. *IEEE Trans. on Information Theory*, 31:302–306, 1985.

26 D. Kowalski. On selection problem in radio networks. In *Proceedings, 24th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 158–166, Las Vegas, NV, USA, 2005. ACM.

27 Dariusz R. Kowalski and Andrzej Pelc. Time of deterministic broadcasting in radio networks with local knowledge. *SIAM Journal on Computing*, 33(4):870–891, 2004. doi:10.1137/S0097539702419339.

28 G. De Marco and G. Stachowiak. Asynchronous shared channel. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 391–400, Washington, DC, USA, 2017. ACM. doi:10.1145/3087801.3087831.

29 Gianluca De Marco. Distributed broadcast in unknown radio networks. *SIAM J. Comput.*, 39(6):2162–2175, 2010. doi:10.1137/080733826.

30 Gianluca De Marco, Tomasz Jurdzinski, and Dariusz R. Kowalski. Optimal channel utilization with limited feedback. In Leszek Antoni Gasieniec, Jesper Jansson, and Christos Levcopoulos, editors, *Fundamentals of Computation Theory - 22nd International Symposium, FCT 2019, Copenhagen, Denmark, August 12-14, 2019, Proceedings*, volume 11651 of *Lecture Notes in Computer Science*, pages 140–152. Springer, 2019. doi:10.1007/978-3-030-25027-0_10.

31 Gianluca De Marco, Tomasz Jurdzinski, Dariusz R. Kowalski, Michal Rózanski, and Grzegorz Stachowiak. Subquadratic non-adaptive threshold group testing. *J. Comput. Syst. Sci.*, 111:42–56, 2020. doi:10.1016/j.jcss.2020.02.002.

32 Gianluca De Marco, Tomasz Jurdzinski, Michal Rózanski, and Grzegorz Stachowiak. Subquadratic non-adaptive threshold group testing. In Ralf Klasing and Marc Zeitoun, editors, *Fundamentals of Computation Theory - 21st International Symposium, FCT 2017, Bordeaux, France, September 11-13, 2017, Proceedings*, volume 10472 of *Lecture Notes in Computer Science*, pages 177–189. Springer, 2017. doi:10.1007/978-3-662-55751-8_15.

33 Gianluca De Marco and Dariusz R. Kowalski. Towards power-sensitive communication on a multiple-access channel. In *2010 International Conference on Distributed Computing Systems, ICDCS 2010, Genova, Italy, June 21-25, 2010*, pages 728–735. IEEE Computer Society, 2010. doi:10.1109/ICDCS.2010.50.

34 Gianluca De Marco and Dariusz R. Kowalski. Contention resolution in a non-synchronized multiple access channel. In *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013*, pages 525–533. IEEE Computer Society, 2013. doi:10.1109/IPDPS.2013.68.

35 Gianluca De Marco and Dariusz R. Kowalski. Contention resolution in a non-synchronized multiple access channel. *Theor. Comput. Sci.*, 689:1–13, 2017. doi:10.1016/j.tcs.2017.05.014.

36 Gianluca De Marco, Dariusz R. Kowalski, and Grzegorz Stachowiak. Brief announcement: Deterministic contention resolution on a shared channel. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, volume 121 of *LIPIcs*, pages 44:1–44:3. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.DISC.2018.44.

37 Gianluca De Marco, Dariusz R. Kowalski, and Grzegorz Stachowiak. Deterministic contention resolution without collision detection: Throughput vs energy. In *41st IEEE International Conference on Distributed Computing Systems, ICDCS 2021, Washington DC, USA, July 7-10, 2021*, pages 1009–1019. IEEE, 2021. doi:10.1109/ICDCS51616.2021.00100.

38 Gianluca De Marco, Marco Pellegrini, and Giovanni Sburlati. Faster deterministic wakeup in multiple access channels. *Discret. Appl. Math.*, 155(8):898–903, 2007. doi:10.1016/j.dam.2006.08.009.
39 Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995. doi:10.1017/CBO9780511814075.
40 B. S. Tsybakov and V. A. Mikhailov. Free synchronous packet access in a broadcast channel with feedback. *Prob. Inf. Transmission*, 14:259–280, 1977.

A Appendix

A.1 Lemma 1

Proof of Lemma 1. Let us consider the first $32qk$ rounds, for some constant $q > 0$, following the round at which the first station wakes up and starts the computation. We will prove that by the end of this interval the wake up has been accomplished whp(k).

Following the algorithm, each awake station, starting from the round at which it wakes up, transmits using the following sequence of probabilities: $q \cdot \frac{1}{2q}$, $q \cdot \frac{1}{2q+1}$, $q \cdot \frac{1}{2q+2}, \ldots$ If we denote by p_i the transmission probability of an arbitrary awake station u at the ith round of its computation, we have that the sum of transmission probabilities of u over the first $32qk$ rounds, after waking up, is

$$s(32qk) = \sum_{i=1}^{32qk} p_i \le q\left(\sum_{i=1}^{32qk} \frac{1}{i}\right) \le q(1 + \ln(32qk)) \ , \qquad (2)$$

where in the last step we have used the right-hand inequality of the following known bounds for the hth partial sum H_h of the harmonic series:

$$\ln(1+h) \le H_h = \sum_{i=1}^{h} \frac{1}{i} \le 1 + \ln h \ . \qquad (3)$$

For any fixed round t, let us consider now the sum of transmission probabilities of all awake stations at time t, denoted as in the previous section by $\sigma(t)$. Since at most k stations can be awake in each round, the average sum $\sigma(t)$ for t ranging over our interval of $32qk$ rounds will be

$$\frac{1}{32qk}\sum_{t=0}^{32qk} \sigma(t) \le \frac{s(32qk)\cdot k}{32qk} \le \frac{q(1+\ln(32qk))\cdot k}{32qk} \le \frac{\ln(32qk)}{16} \ .$$

Of course, in at least half of the interval, $\sigma(t)$ must be not larger than twice the average; therefore there is a set T of at least $16qk$ rounds such that

$$\sigma(t) \le \frac{\ln(32qk)}{8} \ , \qquad (4)$$

for every $t \in T$. Let us consider only rounds in T. We say that a round t is *heavy* when $\sigma(t) > 1/2$ and *light* otherwise. We distinguish two cases.

Case 1: there are at least $8qk$ heavy rounds. Recalling also (4), in at least $8qk$ rounds t, it holds that

$$\frac{1}{2} < \sigma(t) \le \frac{\ln(32qk)}{8} \ . \qquad (5)$$

In [24] (cf. Corollary 5.3.1) it is showed that if (5) holds, than the success probability at round t is at least

$$\left(\frac{1}{16}\right)^{\frac{\ln(32qk)}{8}} \geq \frac{1}{\sqrt{32qk}} .$$

Therefore, the probability that the wake-up does not appear in $8qk$ heavy rounds is at most $(1-1/\sqrt{32qk})^{8qk} = O(1/k^a)$ for an arbitrary constant a depending on q.

Case 2: there are less than $8qk$ heavy rounds. So, there are at least $\delta = 8qk$ light rounds in T. Let $t_1, t_2, \ldots, t_\delta$ be the sequence of consecutive light rounds. In [24] (cf. Claim 1 in the proof of Theorem 10.3) it is showed by induction on i that $\sigma(t_i) \geq q/(2q+i)$, for $1 \leq i \leq \delta$. Consequently,

$$q/(2q+i) \leq \sigma(t_i) \leq 1/2, \text{ for } 1 \leq i \leq \delta.$$

By Corollary 5.3.3 in [24], this implies that the probability of successfully waking up in the ith light round is at least $q/2(2q+i)$. Hence, the probability that the wakeup is not successful is at most

$$\prod_{i=1}^{\delta}(1-\sigma(t_i)) \leq \prod_{i=1}^{\delta}\left(1-\frac{q}{2(2q+i)}\right) \leq \left(\frac{1}{e}\right)^{\sum_{i=1}^{\delta}\frac{q}{2(2q+i)}} \leq \left(\frac{1}{e}\right)^{\frac{q}{2}(\sum_{i=1}^{\delta}\frac{1}{i}-\sum_{i=1}^{2q}\frac{1}{i})}$$
$$= \left(\frac{1}{e}\right)^{\frac{q}{2}(H_\delta - H_{2q})} \leq \left(\frac{1}{e}\right)^{\frac{q}{2}(\ln(1+\delta)-\ln(4q))} = \left(\frac{4q}{1+8qk}\right)^{q/2} \leq \left(\frac{1}{2k}\right)^{q/2} ,$$

where the second last inequality follows by (3) and choosing $q \geq e/2$).

If in the above arguments the contention k is kept but the number of considered rounds is extended by a factor $\log(k'/k)$, the probability of failure in the formulas is raised to $\log(k'/k)$, which results in success whp(k').

We can observe that Protocol 3 gives a maximum number of transmissions per station of $O(\log k)$ whp. This follows from the fact that in expectation the sum of $q/(2q+i)$ over i up to $O(k)$ is $O(\log k)$. The Chernoff bound and then the union bound over the stations give the desired $O(\log k)$ bound whp(k). The same argument gives energy $O(\log k')$ bound whp(k') when the length of the execution is $O(k\log(k'/k))$. ◀

A.2 Lemma 5

In order to prove Lemma 5 we need to show concentration bounds on the lengths of the activity intervals. To this aim, we will use, similarly as in [2], the Azuma-Hoeffding inequality on martingales.

▶ **Definition 15.** *A sequence of random variables $X_0, X_1, \ldots$ is said to be a martingale sequence if for all $i > 0$, $E[X_i|\, X_0, \ldots, X_{i-1}] = X_{i-1}$.*

The following theorem holds (see e.g. [39, p. 92]).

▶ **Theorem 16** (Azuma-Hoeffding inequality)**.** *Let $X_0, X_1, \ldots$ be a martingale sequence such that for each i,*

$$|X_i - X_{i-1}| \leq c_i,$$

where c_i may depend on i. Then, for all $t \geq 0$ and any $\lambda > 0$,

$$\Pr(|X_t - X_0| \geq \lambda) \leq 2\exp\left(-\frac{\lambda^2}{2\sum_{i=1}^{t} c_i^2}\right).$$

Proof of Lemma 5. Fix a sufficiently large positive integer α. Pick any integer j, where $1 \le j \le y$, and consider three cases covering all possible events (i.e., for each j, at least one of the three cases holds):

Case (a): 2^j is at least $\bar{\kappa}^{1/\alpha}$. In such a case, 2^j is a polynomial in $\bar{\kappa}$. Therefore, each ℓ_i for $i \in L_j$ is, by the definition of L_j, upper bounded by $c \cdot 2^j$ whp($\bar{\kappa}^{1/\alpha}$). Since α is a constant, we can set a suitable parameter $\eta > 0$ in the definition of whp such that the upper bound $\ell_i \le c \cdot 2^j$ also holds whp($\bar{\kappa}$). There are at most $\bar{\kappa}$ of such events, by the definition of $\bar{\kappa}$. By applying the union bound to these events, and still choosing a suitable parameter $\eta > 0$ in the definition of whp, we get that $\sum_{i \in L_j} \ell_i$ is $O(2^j w_j)$ holds whp($\bar{\kappa}$).

Case (b): case (a) does not hold and w_j is at least $\bar{\kappa}^{3/\alpha}$. Let $\mathcal{P}$ be the conditional probability space restricted to all situations where the following event $\mathcal{E}$ holds: for all $i \in L_j$, $\ell_i \le c \cdot 2^j \log(\bar{\kappa}/2^j)$. Let $i_1, i_2, \ldots, i_{w_j}$ be any order of the indices of L_j. We can define a sequence of random variables $X_{i_1}, X_{i_2}, \ldots, X_{i_{w_j}}$ such that for $1 \le v \le w_j$, $X_{i_v} = \ell_{i_1} + \cdots + \ell_{i_v} - vc \cdot 2^j \log(\bar{\kappa}/2^j)$.

We can now observe that in the conditional probability space $\mathcal{P}$ the above sequence of random variables is a martingale. Indeed, we have $E[X_i |\, X_0, \ldots, X_{i-1}] = X_{i-1} + E[X_i] = X_{i-1} + E[\ell_{i_1}] + \cdots + E[\ell_{i_v}] - vc \cdot 2^j \log(\bar{\kappa}/2^j) = X_{i-1}$. We have also that $|X_i - X_{i-1}| = |\ell_{i_v} - c \cdot 2^j \log(\bar{\kappa}/2^j)| = O(\bar{\kappa}^{1/\alpha} \log(\bar{\kappa}))$, where the last step follows because in $\mathcal{P}$ we have that $\ell_i \le c \cdot 2^j \log(\bar{\kappa}/2^j)$ holds for all $i \in L_j$ and we are assuming that case (a) does not hold, so $2^j < \bar{\kappa}^{1/\alpha}$. Thus the assumptions of Theorem 16 are satisfied and we can estimate the probability that $\ell_{i_1} + \cdots + \ell_{i_{w_j}}$ is larger than $E[\sum_{i \in L_j} \ell_i] + w_j = O(2^j w_j) + w_j = O(2^j w_j)$. Specifically, the Azuma-Hoeffding inequality in our case implies that this probability, within the conditional probability space $\mathcal{P}$, is at most

$$2 \exp\left(-\frac{w_j^2}{2 \sum_{i=1}^{w_j} O(1)} \right) = 2 \exp\left(-\frac{w_j^2}{2 w_j O(\bar{\kappa}^{1/\alpha} \log(\bar{\kappa}/2^j))} \right).$$

Since w_j is at least $\bar{\kappa}^{3/\alpha}$, the complementary event holds whp($\bar{\kappa}$) in the conditional probability space $\mathcal{P}$.

Recall that $\mathcal{E}$, defining the conditional probability space $\mathcal{P}$, is the event that for all $i \in L_j$, $\ell_i \le c \cdot 2^j \log(\bar{\kappa}/2^j)$. By the definition of L_j, $\ell_i \le c \cdot 2^j \log(\bar{\kappa}/2^j)$ whp ($\bar{\kappa}$). Hence, the probability that event $\mathcal{E}$ does not hold (i.e., the case is outside $\mathcal{P}$) is, by the union bound, at most $O(1/\bar{\kappa})$ for $\alpha > 3$. This implies that $\sum_{i \in L_j} \ell_i \le O(2^j w_j)$ holds whp($\bar{\kappa}$) in the whole sample space.

Case (c): cases (a) and (b) do not hold, i.e., $2^j < \bar{\kappa}^{1/\alpha}$ and $w_j < \bar{\kappa}^{3/\alpha}$, which implies $2^j w_j$ is smaller than $\bar{\kappa}^{4/\alpha}$.

Putting all cases together, the total contribution of j satisfying Cases (a), (b) and (c) is linear in $O(\sum_j 2^j w_j) = O(\bar{\kappa})$ and holds whp($\bar{\kappa}$), by the union bound taken over at most $x \le \bar{\kappa}$ values j and using upper bound $O(2^j w_j)$ that holds whp($\bar{\kappa}$) each. ◀

Smoothed Analysis of Information Spreading in Dynamic Networks

Michael Dinitz ✉
Johns Hopkins University, Baltimore, MD, USA

Jeremy Fineman ✉
Georgetown University, Washington, DC, USA

Seth Gilbert ✉
National University of Singapore, Singapore

Calvin Newport ✉
Georgetown University, Washington, DC, USA

Abstract

The best known solutions for k-message broadcast in dynamic networks of size n require $\Omega(nk)$ rounds. In this paper, we see if these bounds can be improved by smoothed analysis. To do so, we study perhaps the most natural randomized algorithm for disseminating tokens in this setting: at every time step, choose a token to broadcast randomly from the set of tokens you know. We show that with even a small amount of smoothing (i.e., one random edge added per round), this natural strategy solves k-message broadcast in $\tilde{O}(n + k^3)$ rounds, with high probability, beating the best known bounds for $k = o(\sqrt{n})$ and matching the $\Omega(n + k)$ lower bound for static networks for $k = O(n^{1/3})$ (ignoring logarithmic factors). In fact, the main result we show is even stronger and more general: given ℓ-smoothing (i.e., ℓ random edges added per round), this simple strategy terminates in $O(kn^{2/3} \log^{1/3}(n)\ell^{-1/3})$ rounds. We then prove this analysis close to tight with an almost-matching lower bound. To better understand the impact of smoothing on information spreading, we next turn our attention to static networks, proving a tight bound of $\tilde{O}(k\sqrt{n})$ rounds to solve k-message broadcast, which is better than what our strategy can achieve in the dynamic setting. This confirms the intuition that although smoothed analysis reduces the difficulties induced by changing graph structures, it does not eliminate them altogether. Finally, we apply tools developed to support our smoothed analysis to prove an optimal result for k-message broadcast in so-called well-mixed networks in the absence of smoothing. By comparing this result to an existing lower bound for well-mixed networks, we establish a formal separation between oblivious and strongly adaptive adversaries with respect to well-mixed token spreading, partially resolving an open question on the impact of adversary strength on the k-message broadcast problem.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Smoothed Analysis, Dynamic networks, Gossip

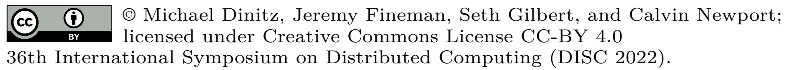
Digital Object Identifier 10.4230/LIPIcs.DISC.2022.18

Related Version *Full Version*: https://arxiv.org/abs/2208.05998 [6]

Funding *Michael Dinitz*: Supported in part by NSF award CCF-1909111.
Jeremy Fineman: Supported in part by NSF grants CCF-1918989 and CCF-2106759.
Seth Gilbert: Supported in part by Singapore MOE grant MOE2018-T2-1-160.

1 Introduction

In this paper, we apply smoothed analysis to the study of k-message broadcast in dynamic networks. We prove that even with a small amount of smoothing, a simple distributed random broadcast strategy can significantly outperform the existing worst-case lower bounds. We then prove that in static networks the complexity of this strategy further improves,



© Michael Dinitz, Jeremy Fineman, Seth Gilbert, and Calvin Newport;
licensed under Creative Commons License CC-BY 4.0

establishing that even in the context of smoothing, changing topologies remain more difficult to move information through than their static counterparts. Finally, we apply the tools developed for these analyses to improve the best-known bounds for k-message broadcast, without smoothing, in the *well-mixed* dynamic network setting. This result is significant in part because when combined with an existing lower bound on well-mixed networks [8], it provides a formal separation between strongly adaptive and oblivious adversaries for k-message broadcast.

1.1 Background

In studying distributed network algorithms, it is common to represent the underlying topology with a graph, where nodes correspond to processes and edges to communication links. In the *dynamic network* setting, these graphs can change from round to round as determined by an adversary. An upper bound proved in a dynamic network is considered strong as it can tolerate the many sources of interference, failure or congestion that alter link availability in real world networks (see [11] for a good review).

Kuhn et al. [12] sparked recent interest in the study of the *k-message broadcast problem*, in which nodes in a network of size n must spread k messages (also called *tokens*) to the whole network. In [12], the results assume the Broadcast CONGEST model in which in each round, each node can broadcast a single bounded-size message, containing at most 1 token. A primary result in the paper is a deterministic algorithm that solves k-message broadcast in $O(nk)$ rounds. For larger values of k, this is notably slower than the $O(n + k)$ rounds required to solve this problem in a static network, underscoring the difficulty of dynamic topologies.

Follow-up work by Dutta et al. [8] proved this result close to optimal with a lower bound that establishes $\Omega(nk/\log n + n)$ rounds are necessary to solve k-message broadcast in this setting. This result is strong in that it holds even for randomized algorithms (with a strongly adaptive adversary), and under the *well-mixed* token assumption in which each token has independent constant probability of starting at each node.

1.2 Key question: is $\tilde{\Omega}(nk)$ fundamental?

Given the importance of information dissemination, an $\tilde{\Omega}(nk)$ lower bound on k-message broadcast is unfortunately strong, especially for large networks attempting to disseminate large amounts of information in a setting with limited bandwidth. Following the approach of Dinitz et al. [7], however, we can investigate whether this bound is fundamental.

In more detail, there are two useful possibilities to consider here. First, this $\tilde{\Omega}(nk)$ bound might be *robust* in the sense that something like nk rounds to broadcast k messages is a natural consequence of network topologies that change. This would be reflected, for example, in the existence of large classes of graphs in which this bound is obviously unavoidable. The second possibility is that the bound is instead *fragile* in the sense that it requires carefully-crafted pathological topologies to induce a complexity of this magnitude, and even small changes to these worst-case graphs enable much more efficient solutions. These distinctions are important because if the $\tilde{\Omega}(nk)$ lower bound due to [8] can be shown to be fragile, this provides hope that more efficient information dissemination can be expected in most real world settings. By contrast, if the bound is robust, this indicates that efficient communication should not be expected in practice.

One approach to distinguishing between robustness and fragility is to apply smoothed analysis. In more detail, in the study of sequential algorithms, Spielman and Teng introduced *smoothed analysis* to help explain why the simplex algorithms works well in practice despite

pessimistic worst-case lower bounds [18, 19]. They proved that the introduction of small random perturbations to otherwise worst-case inputs enabled stronger bounds, indicating the existing lower bound was fragile.

Dinitz et al. [7] subsequently adapted smoothed analysis to the study of distributed algorithms in dynamic networks. In this framework, as in the worst-case setting, an adversary generates an arbitrary dynamic graph to describe the changing network. The individual graphs, however, are then each augmented with ℓ additional random edges, for some smoothing parameter ℓ, before the distributed algorithm in question is run.

For $\ell = 0$, this reduces to the standard worst-case setting where existing lower bounds apply. For $\ell = \binom{n}{2}$, this reduces (more or less) to a random graph setting, in which much stronger upper bounds results are typically possible. As argued in [7], if a worst case lower bound is significantly diminished by smoothed analysis for small ℓ values, then this hints that the original bound is fragile.

The processes and problems studied in [7] were flooding, random walks, and token aggregation. (Follow-up work applied smoothed analysis to the study of the minimum spanning tree [3] and leader election [16] in static graphs.) The k-message broadcast problem features arguably the best-known pessimistic lower bound in the dynamic network setting, but its examination using smoothed analysis was left in [7] as an open problem.

1.3 Our Results

We focus in this paper on *random broadcast*, one of the simplest possible algorithms for disseminating tokens: in each round, each node broadcasts a token chosen uniformly at random from its current token set. This simple strategy will enable us to prove a variety of interesting results on k-message broadcast.

Smoothed analysis of random broadcast in dynamic networks

Applying smoothed analysis as our key tool, we prove that even a small amount of smoothing (i.e., one random edge added per round) is sufficient to enable random broadcast to outperform the worst-case lower bound. This implies both that the existing bound is fragile, and that random broadcast is, in some sense, the *right* strategy for spreading tokens through a dynamic network.

We first establish in Section 5 the baseline result that with no smoothing random broadcast solves the problem in $O(nk)$ rounds, with high probability in n. This matches the deterministic bound from [12].[1] We then investigate the impact of smoothing. In Section 6.2, we show that even with a small amount of smoothing (i.e., $\ell = 1$), random broadcast now terminates in $\tilde{O}(n + k^3)$ rounds, with high probability in n, improving on the best-known $O(nk)$ bound for any $k = \tilde{o}(\sqrt{n})$, and matching the static network lower bound of $\Omega(n + k)$ for $k = \tilde{O}(n^{1/3})$.

We emphasize that 1-smoothing adds at most *one* new edge to the network in each round, which enables at most one extra token dissemination. This smoothing therefore changes the overall bandwidth or connectivity of the graph by only a very small amount.[2] Given that $\Theta(nk)$ rounds might be necessary to solve k-message broadcast, our speed-up in time

[1] Note that [12] gave a deterministic algorithm for the problem, and also explored the problem of termination detection, which further complicates the problem.

[2] Notice, for example, that to significantly increase the conductance or vertex expansion of the graph, you would need to add many more edges.

complexity in this context does not come simply from adding large amounts of extra capacity to a worst-case network: most of the work of token dissemination must still occur over the adversarially specified edges in the network. The smoothing accomplishes something more subtle: as we elaborate in our below discussion of predecessor paths, these extra edges are not eliminating bottlenecks in the underlying dynamic network, but instead providing just enough random noise to allow us to bypass their corresponding potential for congestion.

In reality, our result for $\ell = 1$ is proved as a corollary of our more general result (proved in Section 6.1) showing that random broadcast solves k-message broadcast in $O\left(\frac{kn^{2/3}\log^{1/3} n}{\ell^{1/3}}\right)$ rounds, which for $\ell = 1$ is upper bounded by the $\tilde{O}(n + k^3)$ bound claimed above for all n and k. Notice that in this general form, for $k = o(n^{1/3})$, random broadcast actually *beats* the $\Omega(n + k)$ lower bound for static networks. This is possible because even a small number of additional random edges enables tokens to not only bypass bottlenecks, but also skip ahead in temporal paths, reducing the effective dynamic diameter of the network. As we increase ℓ, we get further improvements. For $\ell = k^3$, for example, we get a sub-linear result, as the increased smoothing both speeds up the rate at which tokens initially spread, and the rate at which they subsequently jump over smoothed edges to locations near their destinations in time and space (see below for elaboration).

Predecessor paths. A key technique in our analysis, presented in Section 4, is the use of graph structures that we call *predecessor paths*, which capture paths that exist over time. They are represented as a sequence of node/round pairs, $(u_1, r_1), (u_2, r_2), \ldots, (u_x, r_x)$, and for each (u_i, r_i), for $i < x$, it is guaranteed that u_i is connected to u_{i+1} during round r_i. We further customize these paths for a given token t, strengthening the guarantee for each (u_i, r_1) such that not only will u_i be connected to u_{i+1} in round r_i, but it will broadcast token t in this round, if it knows it.

For each given destination u_x and token t, therefore, we can reduce the problem of delivering t to u_x to the problem of seeding token t into the appropriate predecessor path. (Intuitively, we are establishing here a net over time and space that can capture a token and then inexorably guide it to the center of the trap.) At a high-level, we can therefore break our smoothed analysis of random broadcast into three phases. During the first phase, we ignore the smoothed edges, and allow the natural dynamics of information spread in these networks spread out each token to a larger set. During the second phase, we allow the smoothed edges to seed these tokens onto the appropriate predecessor paths. During the final phase, the tokens can then traverse these paths to their final destinations.

The lengths of these phases are inter-dependent. Increasing the initial spreading phase, for example, decreases the second phase as now each smoothed edge has a higher probability of selecting a node with a useful token. Similarly, relying on long predecessor paths also reduces the second phase, as now each smoothed edge has more targets to which to deliver a token. Longer predecessor paths, however, necessitate a longer third phase to given tokens time to traverse to their destinations. Our final result balances these dependencies by optimizing the time complexity when we fix all three phases to be the same length.

Lower bound. In Section 6.3, we complement our upper bound analysis of random broadcast with a nearly-matching lower bound. In more detail we describe and analyze a *dynamic star* topology in which the network graph forms a star in each round, but the identity of the center node rotates over time. In this setting, we prove random broadcast's expected time to solve k-message broadcast is in $\tilde{\Omega}\left(\min\left(\frac{kn^{2/3}}{(\ell(k+\ell))^{1/3}}, \frac{kn}{k+\ell}\right)\right)$. This result is approximately

a factor of $(k + \ell)^{1/3}$ below our upper bound analysis, confirming that significantly more efficient analyses are not possible. Notably, this bound establishes the fundamental nature of the drop from n to $n^{2/3}$ in the presence of even a small amount of smoothing.

Smoothed analysis of random broadcast in static networks

A possible interpretation of our upper bounds is the following: "with a small amount of smoothing, dynamic networks behave like static networks". In other words, it might be the case that smoothing removes the differences between dynamic and static networks. In Appendix A, we investigate this issue by studying the behavior of random broadcast in the network toplogies that do not change from round to round. We prove that in the presence of minimum smoothing (i.e., $\ell = 1$) random broadcast completes in static network a polynomial factor of n faster than what is possible in dynamic networks. Formally, we prove that in any static network with 1-smoothing random broadcast completes in $\tilde{O}(k\sqrt{n})$ rounds, with high probability. (Recall, the relevant upper bound result in dynamic networks for 1-smoothing is $\tilde{O}(kn^{2/3})$ rounds.) We then prove this analysis is tight (within logarithmic) terms with a matching lower bound.

At the core of our analysis is a decomposition of an arbitrary static network into at most $O(\sqrt{n})$ components each with diameter at most $O(\sqrt{n})$. We demonstrate that given a collection of t components that know the token, in a single spreading interval of $\tilde{O}(k\sqrt{n})$ rounds, each of the t components is likely to send a given target token over a smoothed edge to a unique new component, effectively doubling the number that now know it. We leverage this doubling behavior to spread a token to a large fraction of the network in only a logarithmic number of spreading intervals.

Well-mixed networks

In [8], the authors introduced the notion of a *well-mixed* network in the context of studying k-message broadcast. They call a network well-mixed if for every node u and token t, node u starts with token t with some independent constant probability. Surprisingly, they observe that their $\Omega(nk)$ lower bound holds even for well-mixed networks. It turns out that even starting with a very uniform token distribution does not make the problem easy.

Accordingly, they replace the broadcast communication model with the much more powerful Symmetric-Diff CONGEST model in which each node can not only send a different token on each outgoing edge, but also perform a set-difference with each of their neighbors before deciding what tokens to send. Given this extra power, they show that it is possible to solve k-message broadcast in a well-mixed network in $\tilde{O}(n + k)$ rounds, with high probability.

Leveraging our predecessor path constructions introduced for our smoothed analysis results, in Appendix B we prove, perhaps surprisingly, that our simple random broadcast algorithm solves k-message broadcast in $O((k/p) \log n)$ rounds, with high probability, where p is the probability that each nodes starts with each token. For the $p = \Theta(1)$ case considered in [8], we strictly improve on the bound they achieved in their more powerful communication model. Indeed, for constant p, our bound is within a single log factor of matching a trivial $\Omega(k)$ lower bound for *all* algorithms in the broadcast communication model.

The key follow-up question, of course, is why our result does not violate the $\Omega(nk)$ lower bound from [8]. The difference is found in the adversary assumptions. The existing lower bound requires a *strongly adaptive* adversary that knows the nodes' random choices in advance and can construct the network topology for a given round based on the knowledge of the tokens nodes are broadcasting in that round. We assume, by contrast, an oblivious adversary that designs the network without advance knowledge of these random bits.

Our well-mixed network result, therefore, opens a clear gap between the strongly adaptive and oblivious adversaries in the context of k-message broadcast. This partially resolves the open question presented in [8] as to whether or not the $\Omega(nk)$ lower bound applies to oblivious adversaries as well.

2 Related Work

Many problems have been studied in various dynamic network models; e.g., [14, 10, 8, 4, 1, 5, 17, 9] (see [11] for a good survey). Interest in k-message broadcast in a dynamic network with broadcast communication was sparked by Kuhn et al. [13], who established the original $O(nk)$ upper bound that provides the baseline for the smoothed analysis deployed in this paper. The relevant matching lower bounds for arbitrary and well-mixed token distributions were subsequently proved by Dutta et al. [8].

Dinitz et al. [7] adapted the smoothed analysis technique, originally introduced by Spielman and Teng [18, 19] in the context of sequential algorithms, to dynamic networks. They studied flooding, random walks and aggregation, and identified k-message broadcast as an important open question. Subsequent work applied this smoothed analysis framework to various other graph problems, including minimum spanning tree construction [3] and leader election [16]. Recently, Meir et al. [15] proposed a variation of graph smoothing, suitable for long-lived processes, in which the smoothing parameter ℓ can be fractional. As noted in [7], smoothed analysis is not the only technique deployed in the literature for sidestepping fragile dynamic network lower bounds. Denysyuk et al. [5], for example, circumvent an exponential lower bound for random walks in dynamic graphs due to [2] by requiring the dynamic graph to include a certain number of static graphs from a well-defined set. In the context of the dynamic radio network model, Ghaffari et al. [9] studied the impact of adversary strength, similarly finding a noticeable gap between oblivious and strongly adaptive adversaries in the context of broadcast.

3 Preliminaries

Here we define the dynamic network models we study and the k-message problem we solve. We also formalize ℓ-smoothing and establish some useful notation and probability results leverage throughout the paper to follow.

Model

We study a dynamic network model in which an execution begins with an oblivious adversary that chooses a *dynamic graph,*, defined as a sequence $\mathcal{G} = G_1, G_2, \ldots$, where each G_i is a connected graph over a common node set V of size $n = |V|$. Time proceeds in synchronous rounds. At the beginning of each round $r \geq 1$, each node $u \in V$ can reliably broadcast a message to is neighbors in G_r. A key difficulty of these models is that u does not know its neighbors in advance.

k-Message Broadcast

The k-message broadcast problem assumes a set T containing $k \geq 1$ unique messages that are also commonly called *tokens* or *rumors*. Each rumor in T starts the execution at one or more nodes. The problem is solved once all nodes have received all k rumors in T. Following the standard convention [12], we assume each node can broadcast at most 1 rumor per round.

ℓ-Smoothing

Fix a dynamic graph $\mathcal{G} = G_1, G_2, \ldots$. Fix a smoothing parameter $\ell \geq 1$. Our goal is to define a smoothing process that adds ℓ random edges to each G_i in $\mathcal{G}$. In an effort to maximize generality, the original definition of ℓ-smoothing from [7] assumed that each G_i was replaced by a graph $\hat{G}_i$ sampled uniformly from the set of graphs that are both "allowable" and within edit distance k of G_i. This was meant to allow both additions and deletions while avoiding illegal topologies (e.g., disconnected graphs).

The results for the Broadcast CONGEST model in [7], however, largely avoided much of this generality, instead applying results (Lemmas 4.1 and 4.2) that establish that this model approximates a simpler model in which edges are randomly added from the set of all edges. For the sake of clarity, in this paper we directly deploy this simpler definition of smoothing.

Formally, after the adversary generates $\mathcal{G}$, we smooth each G_i as follows: (1) randomly generate ℓ edges (with replacement); (2) for each such edge, if it is not already in the smoothed graph we are generating, add it to the graph.

Notation

In the following, we use $\tilde{O}$, $\tilde{\Theta}$, and $\tilde{\Omega}$ to suppress logarithmic factors with respect to n. When we specify a result holds *with high probability*, we mean with failure probability upper bounded by n^{-x} for some sufficiently large constant $x \geq 1$. We also use $[x]$, for integer $x \geq 1$, to indicate the set $\{1, 2, \ldots, x\}$.

Fix an execution of a k-message broadcast algorithm for some token set T of size k and node set V. For each node $u \in V$ and round $r \geq 1$, let $T_u(r)$ be the set of tokens u started with or received by the beginning of round r. We say u *knows* the tokens in $T_u(r)$ at the beginning of round r. Finally, for a given token $t \in T$, let $n_t(r) = |\{u \mid t \in T_u(r)\}|$ be the number of nodes that know token t at the beginning of r.

Useful Probability Results

Many of our high probability results that follow leverage the following useful form of a Chernoff Bound:

▶ **Theorem 1.** *Let $X_1, \ldots, X_j$ be a series of independent random variables such that $X_i \in [0,1]$ where $X = \sum_{i=1}^{j} X_i$ has expectation $E[X] = \mu$. For $\varepsilon \in [0,1]$, $\Pr[X \leq (1-\varepsilon)\cdot\mu] \leq \exp(-(1/2)\cdot\varepsilon^2\mu)$.*

In several places in our analysis, we tame correlated random variables by applying the following stochastic dominance result, which generalizes the above concentration bound. It says that if the probability that the i'th variable is 1 is at least p no matter how the first $i-1$ variables are realized, then we can assume that we have independent Bernoulli variables with parameter p.

▶ **Lemma 2.** *Let $X_1, \ldots, X_j$ be j random variables (not necessarily independent), each of which is distributed over $\{0,1\}$. Suppose there is some $p \in [0,1]$ such that for all $i \in [j]$ and for all $x_1, x_2, \ldots, x_{i-1} \in \{0,1\}$,*

$$\Pr\left[X_i = 1 \mid X_k = x_k \ \forall 1 \leq k < i\right] \geq p.$$

Then

$$\Pr\left[\sum_{i=1}^{j} X_i \leq (1-\varepsilon)pj\right] \leq \exp(-(1/2)\cdot\varepsilon^2 pj)$$

Proof. Consider the following process for sampling random variables $\hat{X}_1, \ldots, \hat{X}_j$. For $i = 1$ to j, do the following. Let $x_1, \ldots, x_{i-1} \in \{0, 1\}$ be the values of $\hat{X}_1, \ldots, \hat{X}_{i-1}$ respectively, and let $q = \Pr[X_i = 1 \mid X_k = x_k \ \forall 1 \le k < i]$. Note that by the assumption of the lemma, $q \ge p$. Sample an independent Bernoulli random variable Y_i which is 1 with probability p and is 0 otherwise. If $Y_i = 1$ then set $\hat{X}_i = 1$. If $Y_i = 0$, then set $\hat{X}_i = 1$ with probability $(q - p)/(1 - p)$ and otherwise set $\hat{X}_i = 0$.

It is easy to see that $\hat{X}_1, \ldots, \hat{X}_j$ and $X_1, \ldots, X_j$ have identical joint distributions. More formally, let $x_i \in \{0, 1\}$ for all $i \in [j]$. Then

$$\begin{aligned}
\Pr[\hat{X}_i = x_i \ \forall i \in [j]] &= \prod_{i=1}^{j} \Pr[\hat{X}_i = x_i \mid \hat{X}_k = x_k \ \forall k < i] \\
&= \prod_{i=1}^{j} \begin{cases} p + (1-p)\frac{\Pr[X_i = 1 \mid X_k = x_k \ \forall 1 \le k < i] - p}{1-p} & \text{if } x_i = 1 \\ (1-p)\frac{1 - \Pr[X_i = 1 \mid X_k = x_k \ \forall 1 \le k < i]}{1-p} & \text{if } x_i = 0 \end{cases} \\
&= \prod_{i=1}^{j} \Pr[X_i = x_i \mid X_k = x_k \ \forall 1 \le k < i] \\
&= \Pr[X_i = x_i \ \forall i \in [j]]
\end{aligned}$$

By the definition of $\hat{X}_i$, we also have the property that $Y_i \le \hat{X}_i$ for all i. Hence Theorem 1 implies that

$$\begin{aligned}
\Pr\left[\sum_{i=1}^{j} X_i \le (1-\varepsilon)pj\right] &= \Pr\left[\sum_{i=1}^{j} \hat{X}_i \le (1-\varepsilon)pj\right] \le \Pr\left[\sum_{i=1}^{j} Y_i \le (1-\varepsilon)pj\right] \\
&\le \exp(-(1/2) \cdot \varepsilon^2 pj)
\end{aligned}$$

as claimed. ◀

4 Random Broadcast Predecessor Paths

Several of the results that follow are built on a structure that we call *predecessor paths*, which are defined with respect to both a given dynamic graph $\mathcal{G}$ and the collection of random bits that determine the choices during a given execution of the random broadcast algorithm. We define and analyze these structures in a general way here. We will later deploy these results to prove specific bounds on random broadcast.

To formally define a predecessor path, we first introduce the notion of a bit assignment $\mathcal{B}$ to be a function $\mathcal{B} : V \times \mathbb{Z}_{>0} \to \{0, 1\}^*$, where $\mathcal{B}(u, r)$ are the random bits node u uses to make its choice of which token to broadcast in round r of running random broadcast. Notice, the combination of a dynamic graph $\mathcal{G}$ and bit assignment $\mathcal{B}$, does not by itself fully specify an execution of random broadcast, as knowledge of the initial token assignment is also required. This information, however, is sufficient for our formal definition.[3]

▶ **Definition 3.** *Fix a dynamic graph $\mathcal{G}$ defined over node set V, token set T, target node $u \in V$, target token $t \in T$, round pair r,r', with $1 \le r < r'$, and bit assignment $\mathcal{B}$. A* predecessor path *$P_{u,t}(r, r')$ for these parameters is a node/round sequence $(u_1, r_1), (u_2, r_2), \ldots, (u_h, r_h)$, where $r \le r_1 < r_2 < \ldots < r_h \le r'$, and $u_i \ne u_j$ for $i, j \in [h]$, $i \ne j$, that satisfies the following with respect to the execution of random broadcast in $\mathcal{G}$ according to bit assignment $\mathcal{B}$:*

[3] A brief aside is that although we call these objects *paths*, the definition actually captures a more general in-tree structure in the time expansion graph.

1. *For each* $i \in [h-1]$*: if* $t \in T_{u_i}(r_i)$*, then* u_i *will broadcast* t *in round* r *and it will be received by at least one node* u_j*, for* $j > i$*.*
2. *If* $t \in T_{u_h}(r_h)$*, then* u_h *will broadcasts* t *during round* r_h *and it will be received by node* u*.*

A natural corollary of this definition is that if *any* node u_i in a predecessor path $P_{u,t}(r, r')$ learns t by r_i, then u will learn t by r'.

Our goal is to describe and analyze a procedure for generating a predecessor path for a given set of parameters. We will then analyze the expected length of the paths created. Roughly speaking, when studying random broadcast, the longer a predecessor path the better, as it gives more opportunities for a given token to arrive at a node that can then send it on its way to the desired destination.

Preliminaries

As discussed, we will be analyzing the simple algorithm in which every node broadcasts a token that it knows uniformly at random. To ease the analysis, though, we assume without loss of generality that (1) the tokens are labelled from 1 to k and that nodes know k; and (2) that a node randomly selects a token to send in a given round by randomly permuting the values from 1 to k and then broadcasting the first token from this sequence that it possesses. Note that this gives the exact same process as the randomized algorithm we care about, which is why this is without loss of generality. It allows us, however, to fix the random process for how a token is chosen even when the available set of tokens to send is unknown.

For a given node set V, token set T, bit assignment $\mathcal{B}$, and round $r \geq 1$, let the *primary token* for u in r, indicated $\delta_u(r)$, be the first token id in u's random permutation for this round as determined by $\mathcal{B}$. In the practical setting where u permutes all values from 1 to $\hat{k}$, then the primary token is the first value in this permutation that correspond to an actual token in T. The important property of a primary token is if $\delta_u(r) = t$, then we know that if $t \in T_u(r)$, node u will broadcast t in this round.

Given a dynamic graph $\mathcal{G} = G_1, G_2, \ldots$, a non-empty node subset $S \subset V$, and a round $r \geq 1$, we further define the *predecessor cut* $c(S, r)$ to be the set of nodes in $V \setminus S$ that neighbor nodes in S in G_r. Notice, because each graph is connected and S is a proper subset of V, these cut partitions are always non-empty.

Predecessor Path Construction

We now describe how to construct a predecessor path for a given set of parameters. This construction process works backwards in time from the end of the desired interval to the beginning. While we describe this process algorithmically, we emphasize that this algorithmic construction is used only in the *analysis* of our algorithms.

In the following, we assume a fixed dynamic network $\mathcal{G} = G_1, G_2, \ldots$, defined over some node set V of size at least 2, a token set T of size $k \geq 1$, and a fixed bit assignment $\mathcal{B}$ for the nodes in V to run random broadcast. We then parameterize the construction process with a node $u \in V$, token $t \in T$, and round range $1 \leq r < r'$. It returns a predecessor path $P_{u,t}(r, r')$ for these parameters.

We now analyze the paths constructed by this procedure, showing establishing the relationship between interval length $(r' - r)$ and path length.

▶ **Theorem 4.** *Fix a dynamic graph* $\mathcal{G}$ *defined over node set* V *of size* $n > 1$*, token set* T *of size* $k \geq 1$*, random bit assignment* $\mathcal{B}$ *for* V*, and error exponent integer* $x > 0$*. For every* $u \in V$*,* $t \in T$*, and rounds* r, r' *where* $r' - r = z \geq 8xk \ln n$*:*

1. *The sequence* $P_{u,t}(r, r')$ *produced by* Path-Construction *is a predecessor path.*
2. *With probability at least* $1 - n^{-x}$*:* $|P_{u,t}(r, r')| > \frac{z}{2k}$*.*

Algorithm 1 Path-Construction(u, t, r, r').

```
P_{u,t}(r, r') ← ϵ;
i ← r';
S ← {u};
while i > r do
    S_{i-1} ← c(S, (i − 1));
    S^{(t)}_{i-1} ← {v | v ∈ S_{i-1} ∧ δ_v(i − 1) = t};
    if |S^{(t)}_{i-1}| > 0 then
        fix any v in S^{(t)}_{i-1};
        S ← S ∪ {v};
        append (v, i − 1) to the front of P_{u,t}(r, r');
    end
    i ← i − 1;
end
return P_{u,t}(r, r')
```

Proof. Fix values for the parameters specified and constrained in the theorem statement. Consider the sequence $P_{u,r}(r, r')$ produced by the Path-Construction algorithm for these parameters. By the definition of this algorithm, $P_{u,t}(r, r')$ is a valid predecessor path. We turn our attention to bounding its size.

At each iteration of the main loop in the procedure, if $|S| < n$, then S_{i-1} is non-empty, as $V \setminus S$ is non-empty and G_{i-1} is connected. Fix some $u \in S_{i-1}$. This node is included into $S^{(t)}_{i-1}$ only if $\delta_v(i-1) = t$, which occurs with probability exactly $1/k$. Therefore, the probability that our path expands in this iteration in the case that $|S| < n$ is at least $1/k$ (it could be larger if there are multiple nodes in S_{i-1}).

For each round $i \in [r, r']$ in the interval, let X_i be the random indicator variable that evaluates to 1 if one of the following conditions holds: (1) $|S| = n$; or (2) $|S| < n$ and $P_{u,t}(r, r')$ grows during the iteration corresponding to round i. Let $Y = \sum_{i \in [r,r']} X_i$. Clearly, Y is an upper bound on the size of $P_{u,t}(r, r')$ returned by the path construction procedure. As we argued that $\Pr(X_i = 1) \geq 1/k$ for each i, we can apply Lemma 2 for these z random variables, $p = 1/k$, and $\varepsilon = 1/2$, to derive $\Pr\left[Y \leq (1/2)\frac{z}{k}\right] \leq \exp\left(-\frac{z}{8k}\right)$. Given our assumption that $z \geq 8xk \ln n$, this error bound is upper bounded by $\exp(-x \ln n) = n^{-x}$, as needed. ◀

5 Random Broadcast in Worst-Case Networks

We begin by showing that in the absence of any additional assumptions, random broadcast matches the best known bound of $O(nk)$ rounds for solving k-message broadcast. The sections that follow will then improve on this baseline. The intuition for this result is straightforward: if token t is not fully spread by round r there is at least one edge over which it could spread with probability at least $1/k$. A union bound and stochastic dominance argument are deployed in the following proof to dispatch dependency and varying token set size issues, respectively.

▶ **Theorem 5.** *Fix a dynamic network $\mathcal{G}$ of size n. Fix a rumor set T of size $k \leq n$. With high probability: random broadcast solves k-message broadcast in $O(nk)$ rounds in $\mathcal{G}$.*

Proof. Fix a token t and round r. If $n_t(r) < n$ then there is at least one edge in the network crossing the cut between nodes that do and nodes that do not know t. Token t is selected for broadcast across this edge with probability at least $1/k$. Let X_r be the random variable that evaluates to 1 under the following two conditions: (1) $n_t(r) = n$; or (2) $n_t(r) < n$ and at least one new node learns t.

Clearly, regardless of the execution, for every r, $\Pr(X_r = 1) \geq 1/k$. Let $Y = \sum_{r=1}^{\alpha nk} X_r$, for a constant $\alpha \geq 1$ that we will fix later. We can apply Lemma 2 for $p = 1/k$, $j = \alpha nk$, and $\varepsilon = 1/2$ to derive that $\Pr[Y \leq (1/2) \cdot (1/k) \cdot (\alpha nk)] \leq \exp\left(-(1/8) \cdot \alpha n\right)$.

Notice, due to the large size of the expectation, for $\alpha \geq 8$, this bound gives us a failure probability exponentially small in n. Clearly then, there is a sufficiently large constant α such that this failure probability is less than or equal n^{-2}, allowing us to apply a union bound over all $k \leq n$ tokens to establish that with high probability: *every* token spreads to all nodes in αnk rounds. ◀

6 Random Broadcast in Smoothed Networks

We now consider the random broadcast algorithm when the initial token distributions and network topologies are arbitrary. In this worst-case setting, as mentioned, the best known k-message broadcast solutions require nk rounds, a bound which is known to be tight within log-factors under certain adversary assumptions. In the previous section, we showed that random broadcast matched this bound as well. The goal of this section is to analyze random broadcast under smoothed analysis.

We show here that if we run the simple random broadcast algorithm on an ℓ-smoothed dynamic network (with $\ell > 0$) then with high probability it solves k-message broadcast after only $O\left(\frac{kn^{2/3}\log^{1/3} n}{\ell^{1/3}}\right)$ rounds. Note that even in the case of little smoothing, e.g., $\ell = 1$, this bound represents nearly an $n^{1/3}$-factor improvement to the round complexity captured by the worst-case bound. To simplify comparison to existing k-message broadcast results, as well as the offline result proved later in this paper, we also prove that for $\ell = 1$, this complexity is upper bounded by $O(n + k^3 \log n)$.

Finally, we note that following the approach of [7], we study only integer ℓ values. As recently demonstrated in [15], fractional smoothing parameters, $0 < \ell < 1$, can also be studied to capture behavior in long-lived networks with slow changes. Our results naturally extend to this case, though we omit this analysis for the sake of clarity.

6.1 Random Broadcast with ℓ-Smoothing

In this section, we look at the case where there are ℓ smoothed edges added per round. Even when $\ell = 1$, we get significant improvements, despite the fact that one edge only enables at most one additional token be transferred per round; thus any non-trivial advantage conveyed by smoothing does not come from directly increasing the capacity of the underlying network. Our goal is to prove the following:

▶ **Theorem 6.** *Fix any dynamic network $\mathcal{G}$ of size n. Fix any rumor set size $k \leq n$. With high probability, random broadcast solves k-message broadcast in $O\left(\frac{kn^{2/3}\log^{1/3} n}{\ell^{1/3}}\right)$ rounds in $\mathcal{G}$ with ℓ-smoothing.*

Our proof proceeds in phases. Note that the algorithm is the same throughout, but our analysis focuses on different quality guarantees in each phase. For the lemmas that follow, assume we have fixed a dynamic graph $\mathcal{G}$, size n, and rumor set T of size k, as specified in the theorem.

6.1.1 Phase #1: Spread

The initial distribution of tokens is arbitrary, and tokens may be located at very few nodes initially. The goal of this first phase is to argue that after enough time tokens are spread out sufficiently across the network, specifically spreading to a δ fraction of nodes. The parameter δ is a function of n and k that we shall set later to balance the length of all phases. We show here that $\Theta(k\delta n)$ rounds suffice to spread all tokens to δn nodes. Notice that if $\delta = 1$, this follows from Theorem 5.

▶ **Lemma 7.** *For any constant $x \geq 1$ and fraction δ with $(1/n)\ln n \leq \delta \leq 1$, there exists a constant $c_1 \geq 1$ such that with probability at least $1 - n^{-x}$: for all $t \in T$, $n_t(R) \geq \delta n$, where $R = c_1 k\delta n$ is the duration of the phase.*

Proof. Fix a given token $t \in T$ and round $r \leq R$. Let X_r be the indicator random variable that evaluates to 1 under two conditions: (1) $n_t(r) \geq \delta n$ (recall that $n_t(r)$ is the number of nodes that know token t at the beginning of round r); or (2) a node learns t for the first time during round r. If the first condition does not hold then not all nodes know t at round r, and hence there is at least one edge connecting a node u that knows t to a node v that does not because the network is assumed to be connected. By the definition of random broadcast, u selects t to broadcast with probability $1/|T_u(r)| \geq 1/k$. It follows that regardless of the execution through the first $r-1$ rounds: $\Pr(X_r = 1) \geq 1/k$.

Let $Y = \sum_{r=1}^{R} X_r$. We can apply our stochastic dominance result (Lemma 2) to $p = 1/k$, $j = R$, and $\varepsilon = 1/2$ to derive the following:

$$\Pr[Y \leq (1/2)\cdot(1/k)\cdot R] \leq \exp\left(-(1/8)\cdot\frac{R}{k}\right) = \exp\left(-(1/8)\cdot c_1\delta n\right)$$
$$\leq \exp\left(-(1/8)\cdot c_1 \ln(n)\right) = n^{-c_1/8}.$$

It is not hard to see that as long as $c_1 \geq 2$, then $Y \geq (1/2)\cdot(1/k)\cdot R$ implies that $n_t(R) \geq \delta n$. This is because otherwise, it must be the case that every X_r which is equal to 1 is because a new node learned token t at round r (not because $n_t(r) \geq \delta n$). But this implies that $n_t(R) \geq Y \geq (c_1/2)\delta n \geq \delta n$.

So if we set $c_1 = 8(x+1)$, we get that the probability that $n_t(R) < \delta n$ is at most $n^{-(x+1)}$. A union bound over all $k \leq n$ tokens provides that $n_t(R) \geq \delta n$ for *every* $t \in T$ with probability at least $1 - n^{-x}$, proving the lemma. ◀

6.1.2 Phase #2: Seed

Now that we have spread each token to $\Omega(\delta n)$ nodes, we next rely on the edges added by smoothing to help sparsely seed these tokens to new random nodes in the network. In the third and final phase we will show that this seeding is likely to have foiled the adversary's attempts to keep certain nodes isolated from certain tokens, and will have instead planted seeds sufficiently close on a temporal path to arrive their destinations.

In more detail, our goal is to show that, for some parameter γ, any sufficiently large set S of size at least $\ln n/\gamma$ will have at least one node in the set receive a given token with high probability during the seed phase. This phase will last for $\Theta((\gamma/\delta)kn)$ rounds following the conclusion of the spread phase.

For example, consider $\delta = 1/k$ and $\gamma = 1/k^2$. Then the length of both the spread phase and the seed phase are $\Theta(n)$ rounds. During the seed phase with these parameters, each node has probability of at least $1/k^2$ of receiving a specific token under consideration. Thus

any set of size $k^2 \ln n$ will receive the token with high probability. Note that these are not the best choices of δ and γ. (With these choices, the total number of rounds including the sink phase is $O(n + k^3 \log n)$ by Lemma 9.)

Our analysis of the seed phase focuses almost entirely on the smoothed edges added to the network topology graph in each round.

▶ **Lemma 8.** *Consider any constant $x \geq 1$ and positive fractions δ and γ. Fix any token $t \in T$, node set $S \subseteq V$ with $|S| \geq (1/\gamma)\ln n$, and round $r_0 \geq 1$ such that $n_t(r_0) \geq \delta n$. With probability at least $1 - n^{-x}$: $|S \cap \{u \mid t \in T_u(r_0 + 2xR)\}| > 0$, where $R = (\gamma/\delta)kn/\ell > k \ln n$ and $2xR$ is the length of the phase.*

Proof. By assumption there are at least δn nodes that know t by the start of this phase (round r_0). If any node in S already knows t at the beginning of this phase, then we are already done. So moving forward, assume no node in S knows t at the start of this phase. Consider the set of potential edges that are useful, denoted E_{useful}, that would connect a node from the initial set that knows t to a node in S. It follows $|E_{useful}| \geq (\delta n)|S| \geq (\delta n)(\ln n/\gamma) = kn^2 \ln(n)/(R\ell)$. We now calculate, for a given round of phase 2, the probability that a smoothed edge selected is from E_{useful}. To do so we leverage the specific definition of smoothing established in Section 3 that treats this as a purely additive process: select a random edge from all possible edges; if it is not already present in the graph, add it to the graph; otherwise leave the graph the same. Therefore, this probability is:

$$\frac{|E_{useful}|}{\binom{n}{2}} > \frac{|E_{useful}|}{n^2} \geq (k \ln n)/(R\ell).$$

As there are ℓ smoothed edges in a round, the probability that none of them are useful is $(1 - k \ln n/(R\ell))^\ell \leq e^{-k \ln n/R} \leq 1 - k \ln n/(2R)$.

We call a round in phase 2 *good* if an edge from E_{useful} is selected *and* the endpoint that knows t selects t to broadcast. Because this latter selection event happens with probability at least $1/k$, we can lower bound the probability of a round being good as $p_{good} \geq \ln n/(2R)$. If a round is not good we call it *bad.* Assume phase 2 runs for $2xR$ rounds for constant $x > 0$. Then the probability that every round is bad is bounded by:

$$(1 - p_{good})^{2xR} \leq \left(1 - \frac{\ln n}{2R}\right)^{2xR} < \exp(-x \ln n) = n^{-x},$$

as required by the lemma statement. ◀

6.1.3 Phase #3: Sink

In the second phase, we leveraged smoothed edges to sparsely seed each token throughout the network in a manner that is independent of the adversary's construction of the dynamic graph. In this final phase, we deploy our predecessor path constructions to show it is likely for each token t and destination node u, that t arrived at an appropriate location in both time and topology to subsequently make its way to u like a flow heading toward a sink (hence the phase name). In particular, we simply need the final phase to be long enough to achieve a predecessor path of size $\Omega(\ln n/\gamma)$. Putting this piece together with the previous phases allows us to prove the following lemma, which almost immediately implies Theorem 6.

▶ **Lemma 9.** *Fix any dynamic network $\mathcal{G}$ of size n. Let δ and γ be any values satisfying $(1/n)\ln n \leq \delta \leq 1$ and $0 < \gamma \leq 1$. Then with high probability, random broadcast completes k message broadcast with ℓ-smoothing in $O(k\delta n + (\gamma/\delta)kn/\ell + k \ln n/\gamma)$ rounds.*

Proof. Our analysis below makes use of a constant $x \geq 1$ that we will fix later. Lemma 7 tells us that there exists a constant $c_1 \geq 1$, such that with probability least $1 - n^{-x}$, for each token $t \in T$, at least δn nodes know t by round $c_1 k \delta n$, for some integer $c_1 > 0$. Let us call this the *spread condition.*

Before we apply the seed phase to each token, we need to identify the sets we are attempting to seed. So we look ahead to the sink phase. Let r_S indicate the first round of the sink phase, which we will calculate later based on the duration of the other two phases. We run this phase for $z = \lceil 8xk \ln n/\gamma \rceil$ rounds. That is, it runs between rounds $r = r_S$ and $r' = r_S + z$. Theorem 4 tells us that with probability at least $1 - n^{-x}$: for every node $u \in V$ and token $t \in T$, the resulting predecessor path $P_{u,t}(r, r')$ is of size greater than $\frac{z}{2k} > \ln n/\gamma$.

This allows to apply our seed phase (Lemma 8) analysis to each such predecessor path. This tells us that so long as the spread condition holds, for each such $S = P_{u,t}(r, r')$, the probability that some node in S receives t during the seed phase is at least $1 - n^{-x}$. The duration of the seed phase is $2x(\gamma/\delta)kn/\ell$ rounds.

Finally, we note that by the definition of a predecessor path, if a node in $P_{u,t}(r, r')$ receives a token t by round r_S, then u receives t by round at most $r_S + z$. We are left to pull together the pieces. To do so, we note that success in solving k-message broadcast by the end of the sink phase requires the following events to occur:

1. The spread condition holds. Call this E_{spread}.
2. For every $u \in V$ and $t \in T$, the predecessor path $P_{u,t}(r, r')$ is sufficient long. Call this $E_{pred}(u, t)$.
3. For each destination node $u \in V$ and token $t \in T$, at least one node in $P_{u,t}(r, r')$ receives t during the seed phase. Call this $E_{seed}(u, t)$.

By our above analysis the probability E_{spread} fails is at most n^{-x}, and the probability $E_{pred}(u, t)$ fails for *any* u and t, as also upper bounded by n^{-x}. For $E_{seed}(u, t)$, the probability of failure for each specific pair u and t, conditioned on E_{spread} and $E_{pred}(u, t)$, is upper bounded by n^{-x}. Therefore, by a union bound, the probability that E_{seed} fails for any such pair is less than $n^{-(x+2)}$. A final union bound then provides that probability any of these events fail is less than $n^{-x} + n^{-x} + n^{-(x+2)} < 1/n$ for a sufficiently large constant x.

Plugging this constant value of x into our above round complexity bounds, and we get that random broadcast succeeds with high probability in $c_1 \delta kn + 2x(\gamma/\delta)kn/\ell + \lceil 8xk \ln n/\gamma \rceil = O(\delta kn + (\gamma/\delta)kn/\ell + k \ln n/\gamma)$ total rounds, as claimed in the theorem. ◀

Proof (of Theorem 6). Choose $\delta = (\log n/(n\ell))^{1/3}$ and $\gamma = (\ell^{1/3})(\log n/n)^{2/3}$. It follows that $\gamma/\delta = (\ell^{2/3})(\log n/n)^{1/3}$. Then by Lemma 9 we get that broadcast completes in at most

$$\begin{aligned} &O((\log n/(n\ell))^{1/3} kn + (\log n/n)^{1/3} kn/\ell^{1/3} + k \log n(n/\log n)^{2/3}/\ell^{1/3}) \\ &= O((kn^{2/3} \log^{1/3} n)/\ell^{1/3}) \end{aligned}$$

rounds, as claimed. ◀

6.2 1-Smoothing

The following result for 1-smoothing is a simple corollary, which allows us to more easily compare to existing results.

▶ **Corollary 10.** *Fix any dynamic network $\mathcal{G}$ of size n. Fix any rumor set size $k \leq n$. With high probability, random broadcast solves k-message broadcast in $O\left(n + k^3 \log n\right)$ rounds in $\mathcal{G}$ with 1-smoothing.*

Proof. Theorem 6 implies that random broadcast takes at most $O(kn^{2/3}\log^{1/3} n)$ rounds. It is easy to see that $kn^{2/3}\log^{1/3} n$ is at most $n + k^3 \log n$ for all values of k. Alternatively, we can set $\delta = 1/k$ and $\gamma = 1/k^2$, and apply Lemma 9 with $\ell = 1$. ◀

6.3 Lower Bound for Random Broadcast in Smoothed Networks

In this section we prove the following theorem.

▶ **Theorem 11.** *For all $n, k, \ell \geq 1$, there are dynamic networks on n nodes and a starting token distribution of k tokens such that random broadcast with ℓ-smoothing has expected completion time of at least $\tilde{\Omega}\left(\min\left(\frac{kn^{2/3}}{(\ell(k+\ell))^{1/3}}, \frac{kn}{k+\ell}\right)\right)$.*

In most "reasonable" regimes (k and ℓ not overwhelmingly large) the minimum in the above lower bound will be achieved by $\frac{kn^{2/3}}{(\ell(k+\ell))^{1/3}}$. Note that this almost matches the upper bound of Theorem 6: it is off by just a $(k+\ell)^{1/3}$ factor. In addition, we also note that when ℓ is large the upper bound *cannot* be tight, while our lower bound can be. To see this, consider the case where $\ell = n$ and $k = o(n)$. For these parameters, essentially every node is the endpoint of an edge added by smoothing in every round, and for each token the probability of broadcasting it is at least $1/k$, and hence for every token the number of nodes who know it will double in at most $O(k)$ rounds. Thus random broadcast will complete after only $O(k \log n)$ rounds. For this case, where $\ell = n$ and $k = o(n)$, our lower bound from Theorem 11 correctly reduces to $\tilde{\Omega}(k)$, while the upper bound from Theorem 6 remains at $\tilde{O}(kn^{1/3})$. This hints that the modest gap between our upper and lower bounds might ultimately be resolved to be closer to the latter result.

6.3.1 Proof of Theorem 11

Our lower bound instance will be the *dynamic star*. The vertices are $v_0, \ldots, v_{n-1}$, and at time $i \in \{1, 2, \ldots, n\}$ the graph will be a star with v_i at the center. Initially node v_0 knows all of the tokens, while nodes $v_1, \ldots, v_{n-1}$ knows all of the tokens *except* for token 1. So random broadcast is complete once all nodes know token 1. Note that this graph is not defined for more than n rounds, but the lower bound that we are trying to prove is at most n due to the second term in the min, so we will not need to consider rounds past n.

Let $t = \min\left(\frac{kn^{2/3}}{(\ell(k+\ell))^{1/3}}, \frac{kn}{k+\ell}\right)/(2000 \log n)$ (we have not optimized the constant or log factors). We will argue that with constant probability, random broadcast has not completed by time t.

Let $A = \{v_i : 1 \leq i \leq t\}$. For $v_i \in A$, we say that v_i is a *good* node if, conditioned on v_i knowing token 1 in round i, v_i will broadcast token 1 in round i (the round where v_i is the center). Let B denote the set of good nodes. Let C_i denote the set of nodes who know token 1 at the beginning of round i. Without smoothing we would have that $C_i \subseteq \{v_0, v_1, \ldots, v_{i-1}\}$, but with smoothing this is not necessarily true. We begin by analyzing C_i under a condition on the good nodes.

▶ **Lemma 12.** *Suppose that for every $i \leq t$, either v_i is not a good node or v_i does not know token 1 at the beginning of round i. Then $|C_i| \leq 100i\left(\frac{k+\ell}{k}\right)\log n$ for all $i \leq t$ with high probability.*

Proof. By assumption, for all $i \leq t$ the center node of the star does not broadcast token 1. Hence in round i, the nodes who learn token 1 consist of at most the center node and some other nodes who learn token 1 via smoothed edges. Even if *all* of the ℓ smoothed edges

had an endpoint in C_i, the expected number of them who transmit token 1 is at most ℓ/k. Hence a Chernoff bound implies that $|C_i| \le |C_{i-1}| + 100 \log n \cdot \left(1 + \frac{\ell}{k}\right)$ with high probability. This high probability allows us to do a union bound over the first t rounds, implying that $|C_i| \le 100i \left(\frac{k+\ell}{k}\right) \log n$ with high probability. ◀

We say that round i has *productive smoothing* if some node in B learns token 1 via an edge added by smoothing. It is easy to see that if after t rounds there has not been any round with productive smoothing, then the assumption in Lemma 12 holds, and hence $|C_i| \le 100i \left(\frac{k+\ell}{k}\right) \log n$ for all $i \le t$ with high probability. Since $t < \frac{nk}{100(k+\ell)\log n}$ this means that not all nodes know token 1 at time t, and so random broadcast has not finished. So we just need to argue that with constant probability, there have been no rounds with productive smoothing before round t.

To see this, first note that by the definition of random broadcast, each node in A is good independently with probability $1/k$. Hence the expected number of good nodes is $|A|/k = t/k$. A standard Chernoff bound then implies that $|B| \le (10t/k) \log n$ with high probability, so from now on we will condition on this being true.

In order for round i to have productive smoothing, at least one of the ℓ edges added by smoothing must have one endpoint in C_i, one endpoint in B, and the endpoint in C_i must choose to broadcast token 1. The probability of this for a single random edge is $\frac{|C_i|}{n} \cdot \frac{|B|}{n} \cdot \frac{1}{k}$, and hence a union bound over all ℓ random edges added in rounded i implies that the probability of productive smoothing in round i is at most

$$\frac{|C_i|}{n} \cdot \frac{|B|}{n} \cdot \frac{\ell}{k} \le \frac{|C_t|}{n} \cdot \frac{10t \log n}{kn} \cdot \frac{\ell}{k} = |C_t| \cdot \frac{10t\ell \log n}{(kn)^2}.$$

If there has been no productive smoothing before round i, then Lemma 12 implies that with high probability $|C_i| \le 100i \left(\frac{k+\ell}{k}\right) \log n$. This is high enough probability for us to take a union bound and still have a high probability bound, so we will assume that *if* there has been no productive smoothing before round i then $|C_i| \le 100i \left(\frac{k+\ell}{k}\right) \log n$.

Let X_i be an indicator random variable for the event that round i has productive smoothing. Then

$$\begin{aligned} \Pr[\exists i \in [t] : X_i = 1] &\le \sum_{i=1}^{t} \Pr\left[X_i = 1 \mid \sum_{j=1}^{i-1} X_j = 0\right] \\ &\le \sum_{i=1}^{t} \left(100t \left(\frac{k+\ell}{k}\right) \log n\right) \left(\frac{10t\ell \log n}{(kn)^2}\right) = \frac{1000t^3\ell\,(k+\ell)\log^2 n}{k^3 n^2} \end{aligned}$$

Since $t \le \frac{kn^{2/3}}{(2000\ell(k+\ell)\log^2 n)^{1/3}}$ then this probability is at most $1/2$, which (as discussed) implies Theorem 11.

References

1 John Augustine, Gopal Pandurangan, Peter Robinson, and Eli Upfal. Towards robust and efficient computation in dynamic peer-to-peer networks. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 2012.

2 Chen Avin, Michal Koucký, and Zvi Lotker. How to explore a fast-changing world (cover time of a simple random walk on evolving graphs). In *Proceedings of the International Colloquium on Automata, Languages and Programming*, 2008.

3 Soumyottam Chatterjee, Gopal Pandurangan, and Nguyen Dinh Pham. Distributed mst: a smoothed analysis. In *Proceedings of the 21st International Conference on Distributed Computing and Networking*, pages 1–10, 2020.

4 Andrea Clementi, Riccardo Silvestri, and Luca Trevisan. Information spreading in dynamic graphs. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 2012.

5 Oksana Denysyuk and Luís Rodrigues. Random walks on evolving graphs with recurring topologies. In *Proceedings of the International Symposium on Distributed Computing*, 2014.

6 Michael Dinitz, Jeremy Fineman, Seth Gilbert, and Calvin Newport. Smoothed analysis of information spreading in dynamic networks, 2022. doi:10.48550/ARXIV.2208.05998.

7 Michael Dinitz, Jeremy T Fineman, Seth Gilbert, and Calvin Newport. Smoothed analysis of dynamic networks. *Distributed Computing*, 31(4):273–287, 2018.

8 Chinmoy Dutta, Gopal Pandurangan, Rajmohan Rajaraman, Zhifeng Sun, and Emanuele Viola. On the complexity of information spreading in dynamic networks. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 717–736. SIAM, 2013.

9 Mohsen Ghaffari, Nancy Lynch, and Calvin Newport. The cost of radio network broadcast for different models of unreliable links. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 2013.

10 Bernhard Haeupler and David Karger. Faster information dissemination in dynamic networks via network coding. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 2011.

11 F. Kuhn and R. Oshman. Dynamic networks: models and algorithms. *ACM SIGACT News*, 42(1):82–96, 2011.

12 Fabian Kuhn, Nancy Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 513–522, 2010.

13 Fabian Kuhn, Nancy Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In *Proceedings of the ACM Symposium on Theory of Computing*, 2010.

14 Fabian Kuhn, Rotem Oshman, and Yoram Moses. Coordinated consensus in dynamic networks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 2011.

15 Uri Meir, Ami Paz, and Gregory Schwartzman. Models of smoothing in dynamic networks. In *Proceedings of the 34th International Symposium on Distributed Computing,*, pages 36:1–36:16, 2020. doi:10.4230/LIPIcs.DISC.2020.36.

16 Anisur Rahaman Molla and Disha Shur. Smoothed analysis of leader election in distributed networks. In *International Symposium on Stabilizing, Safety, and Security of Distributed Systems*, pages 183–198. Springer, 2020.

17 Calvin Newport. Lower bounds for structuring unreliable radio networks. In *Proceedings of the International Symposium on Distributed Computing*, 2014.

18 Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *J. ACM*, 51(3):385–463, 2004.

19 Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis: an attempt to explain the behavior of algorithms in practice. *Commun. ACM*, 52(10):76–84, 2009.

A Random Broadcast in Smoothed Static Networks

As previously argued, a minimum amount of smoothing (i.e., $\ell = 1$) improves the performance of random broadcast in dynamic networks from $O(kn)$ to $\tilde{O}(kn^{2/3})$. To better understand how smoothing supports information spreading, a natural follow-up question is to investigate its impact on random broadcast in *static* networks. If smoothed analysis provides the same bounds for both the dynamic and static settings, this would imply that smoothing essentially bypasses the difficulties induced by changing graph edges. Here we show this is not the case. In more detail, we prove that in static networks, 1-smoothing improves the complexity of random broadcast down to $\tilde{\Theta}(k\sqrt{n})$ rounds, beating what we can guarantee in the dynamic

setting. This establishes a gap between static and dynamic networks with respect to random broadcast, confirming the intuition that network dynamism introduces unique difficulties for information dissemination that smoothing alone cannot fully overcome.

Due to space constraints, we only prove an upper bound of $\tilde{O}(k\sqrt{n})$. The matching lower bound, as well as missing proofs, can be found in the full version [6].

Critical to our analysis is the following graph decomposition result:

▶ **Lemma 13.** *Fix a connected static graph $G = (V, E)$ of size n. There exists a partition of V into components $C_1, C_2, \ldots, C_x$, such that for each i, $1 \le i \le x$: (1) $|C_i| \ge \sqrt{n}$; (2) the subgraph of G induced by C_i is connected and has a diameter at most $6\sqrt{n}$.*

We are now ready to prove our upper bound. The key intuition in the following argument is that we can we analyze the spread of a given target token within the context of the components provided by Lemma 13. We will show, roughly speaking, that when the target token is first spreading, if $\mathcal{A}$ is the set of components that known the target, it is likely that within $\tilde{O}(k\sqrt{n})$ rounds, each component in $\mathcal{A}$ will succeed in seeding the target over a smoothed edge to a unique component not in $\mathcal{A}$, effectively *doubling* the number of components that have learned the token. A logarithmic number of such doublings are sufficient to spread the target to at least half the network, at which point the analysis shifts to the perspective of the remaining components, and argues that within an additional $\tilde{O}(k\sqrt{n})$ rounds, each is likely to connect to an already informed component by a smoothed edge and receive the token. Care is needed in the formal argument to deal with both uneven-sized components and dependencies between the smoothed edge behavior in different components during the same spreading interval.

▶ **Theorem 14.** *Fix some connected static graph G of size n. Fix any rumor set size $k \ge 1$. With high probability, random broadcast solves k-message broadcast in $O(k\sqrt{n}\log^2 n)$ rounds in G with 1-smoothing.*

Proof. Fix a graph $G = (V, E)$ of size n, and a rumor set of size k, as specified by the theorem statement. Fix an arbitrary *target token* from the rumor set. We argue that this target token will spread to full network with the stated complexity.

To do so, we first apply Lemma 13 to partition G into components $C_1, C_2, \ldots, C_x$ with the specified properties. Moving forward, in a given round, we call a component *seeded* if at least one node in the component knows the target token, and call it *completed* if every node knows the target. It is straightforward to establish that once a component is seeded, it will be completed, with high probability, in an additional $O(k\sqrt{n}\log n)$ rounds. One approach to making this argument is to fix a breadth-first search tree in the component rooted at a node that knows the target token. This root will broadcast the target in each round with probability at least $1/k$. Because each broadcast decision is independent, we can apply a Chernoff bound to establish that in $O(k \log n)$ rounds, the root will broadcast the target at least once, with high probability.

We can then repeat this analysis to show that within an additional $O(k \log n)$ rounds, every node at level 1 in the tree will have sent the target token, informing every node at level 2, applying union bounds to ensure that every node at level 1 succeeds with sufficient probability. An additional $O(k \log n)$ rounds moves the token to level 3, and so on. By the guarantees of Lemma 13, the tree has $O(\sqrt{n})$ levels, meaning that $O(k\sqrt{n}\log n)$ rounds is sufficient to spread a target token through the whole component with high probability.

Returning to our main argument, fix a round r. Let $\mathcal{A}_r$ be the subset of components that are seeded at the beginning of round r. Let n_r be the number of nodes in components in $\mathcal{A}_r$. We first consider the case where $n_r < n/2$. To do so, let $\mathcal{B}_r$ be all the components not in

$\mathcal{A}_r$. Our goal is to show that in an additional $O(k\sqrt{n}\log n)$ rounds, the target token will be delivered over smoothed edges to a collection of components in $\mathcal{B}_r$, where they will then spread to a total number of new nodes that is at least a constant fraction of n_r – increasing the number of nodes that know the target token by a constant factor.

To make this argument, we divide this period of $O(k\sqrt{n}\log n)$ rounds starting at round r into three parts. During the first part, we wait for the target token to spread sufficiently to complete every component in $\mathcal{A}_r$. As argued above, with high probability, this takes $O(k\sqrt{n}\log n)$ rounds.

Next we consider a stretch of an additional $O(k\sqrt{n}\log n)$ rounds that we call the *seeding interval*. We will show that during this interval, smoothed edges between components in $\mathcal{A}_r$ and $\mathcal{B}_r$, will seed the target token into a collection of $\mathcal{B}_r$ components whose collective size is a constant fraction of n_r. The final $O(k\sqrt{n}\log n)$ rounds will be dedicated to allowing these seeded $\mathcal{B}_r$ components to complete. (Of course, it is possible that in the time we spent completing components in $\mathcal{A}_r$, the target token might have already made its way to components in $\mathcal{B}_r$ and started spreading, but this only speeds up our efforts.)

We are left then to study closely the behavior of the smoothed edges during the smoothing interval of this period. Though Lemma 13 guarantees that each contains at least $\sqrt{n}$ nodes, it is possible that some might be much larger than this lower limit. It is useful for our purposes to temporarily reorganize these already informed nodes into more uniform-sized groups. With this in mind, let $\mathcal{A}'_r$ be a partition of the nodes in components in $\mathcal{A}_r$ into *groups* that are all of size $\Theta(\sqrt{n})$. For our purposes, it does not matter how this partition is defined. The nodes in each $S \in \mathcal{A}'_r$, for example, do not need to be connected. In the next step of our analysis, we will only concern ourselves with the probability that a node in a given group is selected as an endpoint of a smoothed edge.

Next, fix some arbitrary group $S_1 \in \mathcal{A}'_r$ to consider first. Let $\hat{n}_r$ be the number of nodes in $\mathcal{B}_r$. Because we are still considering the high-level case where $n_r < n/2$, we know $\hat{n}_r \geq n/2$. We now analyze the rounds of the seeding interval in order, stopping at the first round in which: (1) a smoothed edge connects a node $u \in S_1$ to a node in a component in $\mathcal{B}_r$; and (2) u broadcasts the target token. It is straightforward to see that in any given round, a *productive* connection of this type occurs with probability at least $\frac{|S_1|}{n} \cdot \frac{\hat{n}_r}{n} \cdot \frac{1}{k} \geq \frac{1}{2k\sqrt{n}}$.

With high probability, therefore, we will find such a productive connection in a seeding interval of length $O(k\sqrt{n}\log n)$. We now want to consider the other groups in $\mathcal{A}'_r$, and argue that they too will succeed in forming a productive connection during this *same* seeding interval. This will require care to deal properly with dependencies.

The first thing we do is take the component in $\mathcal{B}_r$ at the other end of the productive connection from S_1, and add it to a set $\mathcal{C}$ of successfully seeded components. Before proceeding in our analysis of this seeding interval, we make two checks to see if we are already done. Let n' be the number of nodes in components in $\mathcal{C}$ at this point. If $n_r + n' \geq n/2$, then we can simply wait an additional $O(k\sqrt{n}\log n)$ rounds to complete the component in $\mathcal{C}$, and be done with the high-level case we are considering in which less than half of the nodes know the target token. Similarly, if $n' \geq n_r/2$, then we can finish our analysis of this particular seeding interval as we have accomplished our proximate goal of increasing the number of nodes that know the target token by a constant factor.

If we fail both checks, we must then continue with analyzing our same seeding interval in the hopes of seeding more tokens into $\mathcal{B}_r \setminus \mathcal{C}$. Fix a new group $S_2 \in \mathcal{A}'_r \setminus S_1$. As before, consider rounds in the seeding interval one by one, starting from the first round of the interval, until we arrive at a with a productive connection from S_2 to a not yet seeded component in $\mathcal{B}_r \setminus \mathcal{C}$. In each such round, we argue that the probability of a productive connection is still

in $\Omega(\frac{1}{k\sqrt{n}})$. The main difference as compared to prior groups considered is that the number of nodes that can receive a productive smooth edge decreases as we add more components to $\mathcal{C}$. By our above check, however, we know that the number of nodes in $\mathcal{C}$ is less than $n_r/2 < n/4$. Because we similarly assume that $\mathcal{B}_r$ has at least $n/2$ nodes, then $\mathcal{B}_r \setminus \mathcal{C}$ must still have at least $n/4$ total nodes. The probability of selecting an endpoint in $\mathcal{B}_r \setminus \mathcal{C}$ will therefore always be at least $1/4$, reducing the original productive connection probability calculated for S_1 by only a constant factor for later groups.

As previewed, however, we must also consider dependencies. When considering a given round r' in the seeding interval when considering group S_2, there are two relevant possibilities: (1) r' was the productive connection from our analysis of S_1; (2) r' was not the productive connection for S_1. The first case introduces a problematic dependency, as being productive for one group prevents you from being productive for another. To deal with this dependency we simply *ignore* in our analysis any round that was part of a productive connection for a previously studied group. The second case, by contrast, introduces a useful dependency: knowing that r' was not productive for S_1 only *increases* the probability that it is productive for S_2. A standard negative correlation argument tells us that when lower bounding the probability of a productive connection with respect to S_2, it is fine to treat each round in the second case as succeeding with an independent probability in $\Omega(\frac{1}{k\sqrt{n}})$.

It follows that with high probability, S_2 will also have a productive connection in our seeding interval. At this point, we can repeat the above analysis. Add the newly seeded component to $\mathcal{C}$. Check if we are done. If not, select a new set $S_3 \in \mathcal{A'}_r \setminus \{S_1, S_2\}$ and study the same interval again, round by round, ignoring now only the two rounds in which S_1 and S_2 succeeded in forming productive connections, and so on, until we finally match the criteria for stopping our analysis. (Notice that removing productive rounds from consideration in the seeding interval is not a problem as there are at most $O(\sqrt{n})$ such productive rounds possible given that $|\mathcal{A'}_r| = O(\sqrt{n})$, and our interval length can be made sufficiently long to still provide us the needed high probability of success even with up to $O(\sqrt{n})$ rounds omitted from consideration.)

Moving on with our argument, recall that for our fixed round r, there are two different criteria that might terminate the above seeding analysis. The first is that at least half the nodes in the network learn the target token, at which point we are ready to move on to the second half of our overall analysis, which we will discuss shortly. The second termination criteria is that the number of nodes in components in $\mathcal{B}_r$ that learn the target token is at least a constant factor of n_r. In this case, we fix a new round r' after the seeding interval in question is done, and after all components in $\mathcal{C}$ complete. We then reapply our above analysis starting from r', increasing the number of nodes that know the target node by another constant factor. We can repeat this at most $O(\log n)$ times before at least half the nodes know the target token.

We now consider what happens once we succeed in spreading the target token to at least half the nodes in the network. We can now redeploy pieces of our above spreading argument to show that target token will make it to all remaining nodes in just one more additional spreading interval of length $O(k\sqrt{n}\log n)$ rounds.

We first dispense with the sub-case in which less than $\sqrt{n}$ nodes do not the token; i.e., we are almost done. If this is true, each uninformed node u is within $\sqrt{n}$ of at least one informed node, meaning a straightforward spreading analysis will deliver the token to each such u with high probability within $O(k\sqrt{n}\log n)$ rounds – completing the rumor spreading.

We are left with the sub-case in which somewhere between $\sqrt{n}$ and $n/2$ nodes remain that do not have the target token. Let $\mathcal{A}$ be the set of components that contain at least one node that from this set of nodes that do not know the target. Some of these components are

seeded (i.e., at least one node in them knows the target) and some are not (i.e., no node knows the token). Let $\mathcal{A}'$ be the subset of $\mathcal{A}$ that contains only the non-seeded components. Spend $O(k\sqrt{n}\log n)$ to complete the seeded components from $\mathcal{A}$. We now turn our attention exclusively to the components from $\mathcal{A}'$, as these are the only components at this point which can possibly contain *any* nodes that do not know the target token. We know each such component is of size at least $\sqrt{n}$, and that at least half the total nodes in the network are not in these components. We can therefore apply our above spreading analysis to the components in $\mathcal{A}'$, which establishes that in a single additional spreading interval, every component in $\mathcal{A}'$ will have a productive connection with a completed component, thus seeding it with the target token. We can then complete these components, and therefore complete k-message broadcast, in an additional $O(k\sqrt{n}\log n)$ rounds.

To conclude the proof, we note that all relevant growth and spreading arguments hold with high probability, and that there are at most poly(n) such events that must succeed. This allows us to deploy union bounds to prove that the entire problem is successful, with high probability, after $O(\log n)$ intervals of length $O(k\sqrt{n}\log n)$, yielding the claimed overall time complexity of $O(k\sqrt{n}\log^2 n)$ rounds. ◀

B Random Broadcast in Well-Mixed Networks

Dutta et al. [8] introduced the notion a *well-mixed* network in the context of the k-message broadcast problem. A network satisfies this property if for each token t and node u, with some independent constant probability, u starts with t. The main $\tilde{\Omega}(nk)$ lower bound from [8] also holds for well-mixed networks. To circumvent this bound, they assume a stronger communication model that allows interactive communication on each edge, and provide a randomized algorithm in this setting that solves k-message broadcast in $\tilde{O}(n+k)$ rounds.

Here we deploy our predecessor path constructions, originally designed to support our smoothed analysis, to now explore an alternative non-smoothing method to circumvent the $\tilde{\Omega}(nk)$ lower bound: weakening the adversary. The lower bound in [8] assumes a *strongly adaptive* adversary that knows all the nodes' random bits, allowing it to generate a network graph in each round based on the specific tokens nodes will broadcast during that round. Another natural option is the *oblivious* adversary, in which the adversary generates the graph without advance knowledge of the nodes' random bits. Indeed, the question of whether an oblivious adversary enabled better bounds was identified as important future work in [8].

Using predecessor paths, we prove that with an oblivious adversary, our simple random broadcast strategy solves k-message broadcast in well-mixed networks in $\tilde{O}(k)$ rounds. This improves on the $\tilde{\Omega}(n+k)$ bound from [8], as it eliminates the n dependence. It also matches the trivial $\Omega(k)$ lower bound that holds for *all* k-message broadcast algorithms in a well-mixed network.[4] Indeed, our result is actually more general, showing $\tilde{O}(k/p)$ rounds are needed, when p is the token probability; i.e., the definition of well-mixed in [8] assumes $p = \Theta(1)$.

This result opens a clear separation between strongly adaptive and oblivious adversaries in the context of well-mixed networks, and hints such a separation might exist for arbitrary token distributions as well. It also provide further evidence that the simple random broadcast strategy is a highly effective strategy for information dissemination in these settings.

[4] Fix a static line. Let u be one of the endpoints. With constant probability u is missing $\Omega(k)$ tokens in its initial set. Because u can receive at most one token per round in a line topology, it requires at least $\Omega(k)$ rounds for it to learn these missing tokens one by one.

Formally, we consider the following generalized version of the property concerning initial token distributions introduced in [8]:

▶ **Definition 15.** *Fix a probability $p > 0$. We say an initial token distribution is p-*mixed *if for each node $u \in V$ and token $t \in T$: u's initial token set includes t with independent probability p.*[5]

Given this definition, we prove our main upper bound result:

▶ **Theorem 16.** *Fix a probability $p > 0$. Random broadcast solves k-message broadcast in $O((k/p)\log n)$ rounds, with high probability, when run in a network with a p-mixed token distribution.*

Proof. Fix any u and t. Our first step is to choose a large enough r such that $|P_{u,t}(1, r')| \geq \alpha(1/p)\ln n$, for a constant $\alpha \geq 1$ we will fix later. We want this to hold with probability at least $1 - n^{-4}$. To do so, we can apply Theorem 4 with $x = 4$, $r = 1$, $r' = 32k\alpha(1/p)\ln n$. (Notice, for our parameters, the corresponding $z = r' - r$ value is lower bounded by $8xk\ln n = 32k\ln n$, as required by the theorem statement.) This tells us that with probability at least $1 - n^{-4}$, $|P_{u,t}(1, r')| > \frac{32k\alpha(1/p)\ln n}{2k} \geq \alpha(1/p)\ln n$, as needed. A union bound over the $nk \leq n^2$ possible combinations of processes and tokens, tells us that the probability that any predecessor path is less than this length is less than n^{-2}.

Fix some such path $P_{u,t}(1, r')$ that is sufficiently long; i.e., $q = |P_{u,t}(1, r')| \geq \alpha(1/p)\ln n$. We apply the definition of p-mixed to argue that it is likely that at least one node in this path starts the execution with token t. By definition, of a predecessor path, each node in $P_{u,t}(1, r')$ is unique. By the definition of p-mixed, each u_i starts the execution with token t with independent probability p. Given this independence, we can use the indicator random variable X_i to describe whether or not u_i starts with token t, and then apply our Chernoff Bound form from Theorem 1 to $X = \sum_{i=1}^{r'} X_i$, to bound the probability that X is far from its expectation $\mu = pq$. Formally, we get that $\Pr[Y \leq \mu/2] \leq \exp(-\mu/8)$. Given that $\mu = pq \geq \alpha \ln n$, then for $\alpha \geq 32$, it follows that $\Pr[Y = 0] < n^{-4}$.

A union bound over the $nk < n^2$ node and token pairs tells us that if at all predecessor paths are of length at least q, the the probability *any* path does not contain at least one node starting with the path's target token is less than n^{-2}. A union bound over the failure probabilities for these two events gives us the final result that with high probability, for every $u \in V$ and $t \in T$, at least one node in $P_{u,t}(1, r')$ starts with t. As argued in our previous discussion of predecessor paths, if any node on $P_{u,t}(1, r')$ starts with t then u receives t by round r'. Therefore, with high probability, random broadcast solves k-message broadcast in a p-mixed network in $r' = O((k/p)\log(n))$ rounds, as claimed. ◀

[5] A slight technicality of this definition is that as stated it allows for certain tokens to not show up at all, making termination impossible. A simple fix is to assume that all k tokens are distributed arbitrarily to at least one node, then the random process is deployed to further introduce more tokens into the system.

An Almost Singularly Optimal Asynchronous Distributed MST Algorithm

Fabien Dufoulon
Department of Computer Science, University of Houston, Houston, TX, USA

Shay Kutten
Faculty of Industrial Engineering and Management,
Technion – Israel Institute of Technology, Haifa, Israel

William K. Moses Jr.
Department of Computer Science, University of Houston, Houston, TX, USA

Gopal Pandurangan
Department of Computer Science, University of Houston, Houston, TX, USA

David Peleg
Department of Computer Science and Applied Mathematics,
Weizmann Institute of Science, Rehovot, Israel

Abstract

A singularly (near) optimal distributed algorithm is one that is (near) optimal in *two* criteria, namely, its time and message complexities. For *synchronous* $\mathcal{CONGEST}$ networks, such algorithms are known for fundamental distributed computing problems such as leader election [Kutten et al., JACM 2015] and Minimum Spanning Tree (MST) construction [Pandurangan et al., STOC 2017, Elkin, PODC 2017]. However, it is open whether a singularly (near) optimal bound can be obtained for the MST construction problem in general *asynchronous* $\mathcal{CONGEST}$ networks.

In this paper, we present a randomized distributed MST algorithm that, with high probability, computes an MST in *asynchronous* $\mathcal{CONGEST}$ networks and takes $\tilde{O}(D^{1+\varepsilon} + \sqrt{n})$ time and $\tilde{O}(m)$ messages[1], where n is the number of nodes, m the number of edges, D is the diameter of the network, and $\varepsilon > 0$ is an arbitrarily small constant (both time and message bounds hold with high probability). Since $\tilde{\Omega}(D + \sqrt{n})$ and $\Omega(m)$ are respective time and message lower bounds for distributed MST construction in the standard KT_0 model, our algorithm is message optimal (up to a polylog(n) factor) and almost time optimal (except for a D^{ε} factor). Our result answers an open question raised in Mashregi and King [DISC 2019] by giving the first known asynchronous MST algorithm that has sublinear time (for all $D = O(n^{1-\varepsilon})$) and uses $\tilde{O}(m)$ messages. Using a result of Mashregi and King [DISC 2019], this also yields the first asynchronous MST algorithm that is sublinear in both time and messages in the KT_1 $\mathcal{CONGEST}$ model.

A key tool in our algorithm is the construction of a low diameter rooted spanning tree in asynchronous $\mathcal{CONGEST}$ that has depth $\tilde{O}(D^{1+\varepsilon})$ (for an arbitrarily small constant $\varepsilon > 0$) in $\tilde{O}(D^{1+\varepsilon})$ time and $\tilde{O}(m)$ messages. To the best of our knowledge, this is the first such construction that is almost singularly optimal in the asynchronous setting. This tree construction may be of independent interest as it can also be used for efficiently performing basic tasks such as *verified* broadcast and convergecast in asynchronous networks.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Mathematics of computing → Probabilistic algorithms; Mathematics of computing → Discrete mathematics

Keywords and phrases Asynchronous networks, Minimum Spanning Tree, Distributed Algorithm, Singularly Optimal

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.19

Related Version *Full Version*: `https://arxiv.org/abs/2210.01173`

[1] The $\tilde{O}$ notation hides a polylog(n) factor and the $\tilde{\Omega}$ notation hides a $1/\text{polylog}(n)$ factor.

© Fabien Dufoulon, Shay Kutten, William K. Moses Jr., Gopal Pandurangan, and David Peleg; licensed under Creative Commons License CC-BY 4.0

Funding *Fabien Dufoulon*: This work was supported in part by NSF grants CCF-1717075, CCF-1540512, IIS-1633720, and BSF grant 2016419.
Shay Kutten: This work was supported in part by the Bi-national Science Foundation (BSF) grant 2016419 and supported in part by ISF grant 1346/22.
William K. Moses Jr.: This work was supported in part by NSF grants CCF1540512, IIS-1633720, CCF-1717075, and BSF grant 2016419.
Gopal Pandurangan: This work was supported in part by NSF grants CCF-1717075, CCF-1540512, IIS-1633720, and BSF grant 2016419.
David Peleg: This work was supported in part by the US-Israel Binational Science Foundation grant 2018043.

1 Introduction

1.1 Background and Motivation

Singularly (near) optimal distributed algorithms are those that are (near) optimal both in their message complexity and in their time complexity.[2] The current paper is intended as a step in expanding the study of "which problems admit singularly optimal algorithms" from the realm of synchronous $\mathcal{CONGEST}$ networks to that of *asynchronous* ones.

An important example of a problem that has been studied in the context of singularly (near) optimal algorithms is minimum-weight spanning tree (MST) construction. This has become a rather canonical problem in the sub area of distributed graph algorithms and was used to demonstrate and study various concepts such as the congested clique model (Lotker et al. [40]), proof labeling schemes (Korman et al. [36]), networks with latency and capacity (Augustine et al. [3]), cognitive radio networks (Rohilla et al. [52]), distributed applications of graph sketches (King et al. [33]), distributed computing with advice (Fraigniaud et al. [21]), distributed verification and hardness of approximation (Kor et al. [34], Korman and Kutten [35] and Das Sarma et al. [14]), self-stabilizing algorithms (Gupta and Srimani [29] and many other papers), distributed quantum computing (Elkin et al. [19]) and more. The study of the MST problem in what we now call the $\mathcal{CONGEST}$ model started more than forty years ago, see Dalal, and also Spira [12, 13, 56].

The seminal paper of Gallager, Humblet, and Spira (GHS) [22] presented a distributed algorithm for an *asynchronous* network that constructs an MST in $O(n \log n)$ time using $O(m + n \log n)$ messages, where n and m denote the number of nodes and the number of edges of the network, respectively. The time complexity was later improved by Awerbuch and by Faloutsos and Moelle to $O(n)$ [5, 20], while keeping the same order of message complexity.

The message complexity of GHS algorithm is (essentially) optimal, since it can be shown that for any $1 \leq m \leq n^2$, there exists a graph with $\Theta(m)$ edges such that $\Omega(m)$ is a lower bound on the message complexity of constructing even a spanning tree (even for randomized algorithms) [38].[3] Moreover, the time complexity bound of $O(n)$ bound is *existentially* optimal (in the sense that there exist graphs (of high diameter) for which this is the best possible). However, the time bound is not optimal if one parameterizes the running time in terms of the network diameter D, which can be much smaller than n. In a *synchronous* network, Garay, Kutten, and Peleg [23] gave the first such distributed algorithm for the MST

[2] In this paper, henceforth, when we say "near optimal" we mean "optimal up to a polylog(n) factor", where n is the network size.
[3] This message lower bound holds in the so-called KT_0 model, which is assumed in this paper. See Section 1.4 for more details.

problem with running time $\tilde{O}(D+n^{0.614})$, which was later improved by Kutten and Peleg [39] to $\tilde{O}(D+\sqrt{n})$ (again for a synchronous network).However, both these algorithms are not message-optimal as they exchange $O(m+n^{1.614})$ and $O(m+n^{1.5})$ messages, respectively.

Conversely, it was established by Peleg and Rubinovich [51] that $\tilde{\Omega}(D+\sqrt{n})$ is a lower bound on the time complexity of distributed MST construction that applies even to low-diameter networks ($D=\Omega(\log n)$), and to the synchronous setting. The lower bound of Peleg and Rubinovich applies to exact, deterministic algorithms. This lower bound was further extended to randomized (Monte Carlo) algorithms, approximate constructions, MST verification, and more (see [41, 40, 17, 14]).

Pandurangan, Robinson and Scquizzato [47, 49] showed that MST admits a randomized singularly near optimal algorithm in *synchronous* $\mathcal{CONGEST}$ networks; their algorithm uses $\tilde{O}(m)$ messages and $\tilde{O}(D+\sqrt{n})$ rounds. Subsequently, Elkin [18] presented a simpler, singularly optimal deterministic MST algorithm, again for synchronous networks.

For *asynchronous* networks, one can obtain algorithms that are separately time optimal (by combining [39] with a synchronizer, see Awerbuch [4]) or message optimal [22] for the MST problem, but it is open whether one can obtain an asynchronous distributed MST algorithm that is singularly (near) optimal. This is one of the main motivations for this work. An additional motivation is to design tools that can be useful for constructing singularly optimal algorithms for other fundamental problems in asynchronous networks.

In general, designing singularly optimal algorithms for *asynchronous* networks seems harder compared to synchronous networks. In *synchronous* networks, besides MST construction, singularly (near) optimal algorithms have been shown in recent years for leader election, (approximate) shortest paths, and several other problems [38, 30]. However, all these results *do not* apply to asynchronous networks. Converting synchronous algorithms to work on asynchronous networks generally incur heavy cost overhead, increasing either time or message complexity or both substantially. In particular, using *synchronizers* [4] to convert a singularly optimal algorithm to work in an asynchronous network generally renders the asynchronous algorithm not singularly optimal. Using a synchronizer can significantly increase either the time or the message complexity or both far beyond the complexities of the algorithm presented here. Furthermore, there can be a non-trivial cost associated with *constructing* such a synchronizer in the first place.

For example, applying the simple α synchronizer [4] (which does not require the a priori existence of a leader or a spanning tree) to the singularly optimal synchronous MST algorithm of [47, 49] or [18] yields an asynchronous algorithm with message complexity of $\tilde{O}(m(D+\sqrt{n}))$ and time complexity of $\tilde{O}(D+\sqrt{n})$; this algorithm is time optimal, but *not* message optimal. Some other synchronizers (see, e.g., Awerbuch and Peleg [9]), do construct efficient synchronizers that can achieve near optimal conversion from synchronous to asynchronous algorithms with respect to both time and messages, but constructing the synchronizer itself requires a substantial preprocessing or initialization cost. For example, the message cost of the synchronizer setup protocol of [9] can be as high as $O(mn)$.

Another rather tempting idea to derive an MST algorithm that would be efficient both in time and in messages would be to convert a result of Mashreghi and King [44] (see also [43] and discussion in Section 1.4), originally designed in the asynchronous KT_1 $\mathcal{CONGEST}$ model[4] to the more common KT_0 model assumed here. In particular, they give an asynchronous MST algorithm that takes $O(n)$ time and $\tilde{O}(n^{1.5})$ messages. Note that one can convert an

[4] In KT_1 model it is assumed that nodes know the identities of their neighbors (cf. Section 1.4), unlike the KT_0 model, where nodes don't have that knowledge.

algorithm in the KT_1 model to work in the KT_0 model by allowing each node to communicate with all its neighbors in one round; this takes an additional $\tilde{O}(m)$ messages. Hence, with such a conversion the message complexity of the above algorithm would be essentially optimal (i.e., $\tilde{O}(m)$), but the time complexity would be $O(n)$ which is only existentially optimal, and can be significantly higher than the lower bound of $\tilde{O}(D + \sqrt{n})$. In fact, as we will discuss later, our result answers an open question posed in [44] and gives MST algorithms with improved bounds in asynchronous KT_1 model (cf. Section 1.3).

Instead of using a synchronizer, a better approach might be to design an algorithm directly for an asynchronous network. As an example, consider the fundamental leader election problem, which is simpler than the MST construction problem. Till recently, a singularly optimal asynchronous leader election algorithm was not known. Applying a synchronizer to known *synchronous* singularly optimal leader election algorithms *does not* yield singularly optimal asynchronous algorithms. For example, applying the simple α synchronizer to the singularly optimal synchronous leader election algorithm of [38] yields an asynchronous algorithm with message complexity of $O(mD \log n)$ and time complexity of $O(D)$; this algorithm is not message optimal, especially for large diameter networks. Other synchronizers such as β and γ of [4] and that of [9], require the a priori existence of a *leader* or *a spanning tree* and hence cannot be used for leader election. The work of Kutten et al. [37] presented a singularly (near) optimal leader election for asynchronous networks that takes $\tilde{O}(m)$ messages and $\tilde{O}(D)$ time.[5] That algorithm did not use a synchronizer and was directly designed for an asynchronous network. The leader election algorithm of [37] is a useful subroutine in our MST algorithm.

1.2 The Distributed Computing Model

The distributed network is modeled as an arbitrary undirected connected weighted graph $G = (V, E, w)$, where the node set V represent the processors, the edge set E represents the communication links between them, and $w(e)$ is the weight of edge $e \in E$. D denotes the hop-diameter (that is, the unweighted diameter) of G, in this paper, diameter always means hop-diameter. We also assume that the weights of the edges of the graph are all distinct. This implies that the MST of the graph is unique. (The definitions and the results generalize readily to the case where the weights are not necessarily distinct.) We make the common assumption that each node has a unique identity (this is not essential, but simplifies presentation), and at the beginning of computation, each node v accepts as input its own identity number (ID) and the weights of the edges incident to it. Thus, a node has only *local* knowledge. We assume that each node has ports (each port having a unique port number); each incident edge is connected to one distinct port. A node *does not* have any initial knowledge of the other endpoint of its incident edge (the identity of the node it is connected to or the port number that it is connected to). This model is referred to as the *clean network model* in [50] and is also sometimes referred to as the KT_0 model, i.e., the initial (K)nowledge of all nodes is restricted (T)ill radius 0 (i.e., just the local knowledge) [50]. The KT_0 model is extensively used in distributed computing literature including MST algorithms (see e.g., [50, 48] and the references therein). While we design an algorithm for the KT_0 model, our algorithm also yields an improvement in the KT_1 model [7, 50] where each node has an initial knowledge of the identities of its neighbors.

[5] This algorithm is singularly near optimal, since $\Omega(m)$ and $\Omega(D)$ are message and lower bounds for leader election even for randomized Monte Carlo algorithms [38].

We assume that nodes have knowledge of n (in fact a constant factor approximation of n is sufficient), the network size. We note that quite a few prior distributed algorithms require knowledge of n, see e.g. [6, 53, 2, 37]. We assume that processors can access *private unbiased random bits*.

We assume the standard *asynchronous $\mathcal{CONGEST}$* communication model [50], where messages (each message is of $O(\log n)$ bits) sent over an edge incur unpredictable but finite delays, in an error-free and FIFO manner (i.e., messages will arrive in sequence). As is standard, it is assumed that a message takes *at most one time unit* to be delivered across an edge. Note that this is just for the sake of the analysis of time complexity, and does not imply that nodes know an upper bound on the delay of any message. As usual, local computation within a node is assumed to be instantaneous and free; however, our algorithm will involve only lightweight local computations.

We assume an *adversarial wake-up* model, where node wake-up times are scheduled by an adversary (who may decide to keep some nodes dormant) which is standard in prior asynchronous protocols (see [1, 22, 55]). Nodes are initially asleep, and a node enters the execution when it is woken up by the environment or upon receiving messages from other nodes.[6]

The time complexity is measured from the moment the first node wakes up. The adversary wakes up nodes and delays each message in an *adaptive* fashion, i.e., when the adversary makes a decision to wake up a node or delay a message, it has access to the results of all previous coin flips. In the asynchronous setting, once a node enters execution, it performs all the computations required of it by the algorithm, and sends out messages to neighbors as specified by the algorithm. At the end of the computation, we require each node to know which of its incident edges belong to the MST. When we say that an algorithm has termination detection, we mean that all nodes detect termination, i.e., each node detects that its own participation in the algorithm is over.

1.3 Our Contributions

Almost Singularly Optimal Asynchronous MST Algorithm. Our main contribution is a randomized distributed MST algorithm that, with high probability, computes an MST in *asynchronous $\mathcal{CONGEST}$* networks and takes $\tilde{O}(D^{1+\varepsilon} + \sqrt{n})$ time and $\tilde{O}(m)$ messages, where n is the number of nodes, m the number of edges, D is the diameter of the network, and $\varepsilon > 0$ is an arbitrarily small constant (both time and message bounds hold with high probability) (cf. Theorem 9). Since $\tilde{\Omega}(D + \sqrt{n})$ and $\Omega(m)$ are respective time and message lower bounds for distributed MST construction in the KT_0 model, our algorithm is message optimal (up to a polylog(n) factor) and almost time optimal (except for a $\tilde{O}(D^{\varepsilon})$ factor).

Asynchronous MST in KT_1 in Sublinear Messages and Time. Our result answers an open problem raised in Mashregi and King [44] (see also [45, 43]). They ask if there exists an asynchronous MST algorithm that takes sublinear time if the diameter of the network is low, and has $\tilde{O}(m)$ message complexity. They remark that if such an algorithm exists, then it would improve their result giving better bounds for asynchronous MST in KT_1

[6] Although standard, the adversarial wake up model, in our setting, is not more difficult compared to the alternative *simultaneous wake up* model where all nodes are assumed to be awake at the beginning of the computation. Indeed, in the adversarial wake up model, awake nodes can broadcast (by simply flooding) a "wake up" message which can wake up all nodes; this takes only $O(m)$ messages and $O(D)$ time and hence within the singularly optimal bounds.

$\mathcal{CONGEST}$. Our result answers their question in the affirmative by giving the first known asynchronous MST algorithm that has sublinear time (for all $D = O(n^{1-\delta})$, where $\delta > 0$ is an arbitrarily small constant) and uses $\tilde{O}(m)$ messages. Furthermore, as indicated in Mashregi and King [44], this also yields the first asynchronous MST algorithm that is *sublinear* in *both* time and messages in the KT_1 $\mathcal{CONGEST}$ model. More precisely, plugging our asynchronous MST algorithm in the result of [44]([Theorem 1.2]) gives an asynchronous MST algorithm that takes $\tilde{O}(D^{1+\varepsilon} + n^{1-2\delta})$ time and $\tilde{O}(n^{3/2+\delta})$ messages for any small constant $\varepsilon > 0$ and for any $\delta \in [0, 0.25]$ (cf. Theorem 10). This gives a tradeoff result between time and messages. In particular, setting $\delta = 0.25$ yields an asynchronous MST algorithm that has (almost optimal) time complexity $\tilde{O}(D^{1+\varepsilon} + \sqrt{n})$ and message complexity $\tilde{O}(n^{7/4})$.

Low Diameter Spanning Tree Construction. A key tool in our algorithm is the construction of a low diameter rooted spanning tree in asynchronous $\mathcal{CONGEST}$ that has depth $\tilde{O}(D^{1+\varepsilon})$ (for an arbitrarily small constant $\varepsilon > 0$) in time $\tilde{O}(D^{1+\varepsilon})$ time and $\tilde{O}(m)$ messages. To the best of our knowledge, this is the first such construction that is almost singularly optimal in the asynchronous setting. This tree construction is of independent interest as it can also be used for *efficiently* (under both time and messages) performing tasks such as upcast and downcast which are very common tools in distributed algorithms (these are described, for completeness, in Appendix A). Informally, an upcast (using the tree) provides a feedback (i.e., verification) to the broadcast (downcast) initiator such that (1) the broadcast initiator knows when the broadcast terminates (based on acknowledgements from all nodes) and (2) the initiator can get compute a value based on the inputs of all the nodes (e.g., their sum). This verified broadcast is crucial in the asynchronous setting that allows the initiator to know when the broadcast has reached all nodes and thereafter proceed to the next step of the computation.

We note that one could have used a BFS tree instead of a low-diameter tree. However, the best known BFS tree construction in the asynchronous setting is due to Awerbuch [6] which takes $O(D^{1+\varepsilon})$ time and $O(m^{1+\varepsilon})$ messages (for arbitrarily small constant $\varepsilon > 0$). This algorithm (which is deterministic) is not message optimal, unlike ours, and hence will only yield an MST algorithm with $O(m^{1+\varepsilon})$ message complexity. Furthermore, though our algorithm does not compute a BFS (but it is sufficient for MST purposes) and is randomized, it is significantly simpler to understand and prove correctness for when compared to Awerbuch's algorithm. We also note that apart from the leader election and spanning tree primitives, the rest of the MST algorithm is deterministic.

1.4 Additional Related Work

The distributed MST problem has been studied intensively for the last four decades and there are several results known in the literature, including several recent results, both for synchronous and asynchronous networks (including the ones mentioned in Section 1), see e.g., [18, 16, 48, 24, 30, 32, 45, 42, 47, 49] and the references therein.

We note that the results of this paper and that of leader election of [37] (for asynchronous networks) as well as those of [47, 49] and [18] (for synchronous networks) assume the so-called *clean network model*, a.k.a. KT_0 [50] (see Section 1.2), where nodes do not have initial knowledge of the identity of their neighbors. But the optimality of above results does not in general apply to the KT_1 model, where nodes have initial knowledge of the identities of their neighbors. It is clear that for time complexity by itself, the distinction between KT_0 and KT_1 does not matter (as one can simulate KT_1 in KT_0 in one round/time unit by each node sending its ID to all its neighbors) but it is significant when considering message complexity

(as the just mentioned simulation costs $\Theta(m)$ messages). Awerbuch et al. [7] show that $\Omega(m)$ is a message lower bound for broadcast (and hence for construction of a spanning tree as well) in the KT_1 model, if one allows only (possibly randomized Monte Carlo) comparison-based algorithms, i.e., algorithms that can operate on IDs only by comparing them. (We note that all algorithms mentioned earlier in this subsection are comparison-based, including ours.)

On the other hand, for *randomized non-comparison-based* algorithms, the message lower bound of $\Omega(m)$ does not apply in the KT_1 model. King et al. [33] presented a randomized, non-comparison-based Monte Carlo algorithm in the KT_1 model for MST construction in $\tilde{O}(n)$ messages ($\Omega(n)$ is a message lower bound) (see also [42]). While this algorithm achieves $o(m)$ message complexity (when $m = \omega(n \operatorname{polylog} n)$), it is *not* time-optimal, as it takes time $\tilde{O}(n)$ rather than $\tilde{O}(D + \sqrt{n})$. Algorithms with improved round complexity but worse message complexity, and more generally, trade-offs between time and messages, are shown in [26, 27]. We note that all these results are for *synchronous* networks. As discussed in Section 1, the works of [44, 43, 45] address asynchronous MST construction in KT_1 model and present algorithms that take $o(m)$ messages.

2 Low Diameter Spanning Tree Algorithm

Let us now describe a novel algorithm for constructing a low diameter spanning tree in a time-efficient and (near) message-optimal manner in an *asynchronous* network. This serves as a crucial ingredient for our MST algorithm of Section 3.

2.1 Randomized Low Diameter Decomposition (MPX)

Let $\overline{G} = (\overline{V}, \overline{E})$ be any (undirected, unweighted) graph with $\overline{n} \leq n$ nodes and $\overline{m} \leq m$ edges; in particular, $\overline{G}$ can be different from the communication graph. A probabilistic (β, r) *low diameter decomposition* of $\overline{G}$ is a partition of $\overline{V}$ into disjoint node sets $\overline{V}_1, \dots, \overline{V}_t$ called *clusters*. The partition satisfies (1) each cluster $\overline{V}_i$ has strong diameter r, i.e., $dist_{\overline{G}[V_i]}(u, v) \leq r$ for any two nodes $u, v \in \overline{V}_i$, and (2) the probability that an edge $e \in \overline{E}$ is an inter-cluster edge (that is, the endpoints of e are in different clusters) is at most β.

MPX Decomposition in Synchronous $\mathcal{CONGEST}$. Let us describe a simple distributed variant of the MPX decomposition algorithm of Miller et al. [46] – Procedure **MPX** – executed in a synchronous setting with simultaneous wakeup on graph $\overline{G}$. In Subsect. 2.2, we execute the algorithm on virtual cluster graphs (where each node is in fact a set of nodes in the communication graph G) and also describe the distributed simulation required to do so.

Let $\delta_{max} = \lfloor 2 \cdot \frac{\ln n}{\beta} \rfloor$. Initially, each node $v \in \overline{V}$ draws a random variable δ_v from the exponential random distribution with parameter β and sets its *start-time* variable S_v to $\max\{1, \delta_{max} - \lfloor \delta_v \rfloor\}$. Procedure **MPX** guarantees the following through simple flooding: (1) each node $v \in \overline{V}$ is assigned to the cluster of the node $u = argmin_{w \in \overline{V}}\{(dist_{\overline{G}}(v, w) + S_w, id_w)\}$ and (2) each cluster has a spanning tree of depth at most δ_{max}. (Each node locally keeps information about the edge to its parent in the spanning tree. In other words, the spanning tree is oriented towards the root.)

More precisely, the "simple flooding" is done in $\delta_{max} + 1$ rounds. Initially, all nodes are *unassigned*. In round i, each newly-assigned node v (i.e., assigned in round $i - 1$) sends to its neighbors a message containing the ID of the cluster leader. Other assigned nodes do nothing. Finally, for each unassigned node v, let M_{id} be the set containing all received IDs, as well as id_v if $S_v = i$. If M_{id} is the empty set, v does nothing. Otherwise, v assigns itself to the cluster of the node u with the lexicographically smallest ID in M_{id}. If $u \neq v$, v keeps

the edge (an arbitrary one if there are multiple such edges) along which it receives id_u as the edge to its parent. (Note that this spanning tree guarantees that the cluster is connected and has strong diameter at most $\frac{4\ln n}{\beta}$.)

Analysis. The following lemmas are known results from [46, 31, 10, 11]. For completeness, proofs are given in Appendix B.

▶ **Lemma 1.** *Procedure **MPX** computes a* $(2\beta, \frac{4\ln n}{\beta})$ *low-diameter decomposition of* $\overline{G}$ *w.h.p. in* $O(\frac{\ln n}{\beta})$ *time and* $O(m\frac{\ln n}{\beta})$ *messages in the synchronous setting.*

From the low diameter decomposition computed by Procedure **MPX** (or in fact, from any partition $\mathcal{P}$ of $\overline{V}$ into disjoint node sets $\overline{V}_1, \ldots, \overline{V}_t$), one can define a cluster graph $\overline{G}^* = (\overline{V}^*, \overline{E}^*)$, as follows. Its node set $\overline{V}^* = \{\overline{V}_1, \ldots, \overline{V}_t\}$ consists of cluster nodes, one for each cluster $\overline{V}_i$ of the decomposition, and two cluster nodes $\overline{V}_i$ and $\overline{V}_j$ are adjacent in $\overline{G}^*$ if there exist two nodes w, w' in $\overline{V}$ such that $w \in \overline{V}_i$, $w' \in \overline{V}_j$ and $(w, w') \in \overline{E}$. We call $\overline{G}^*$ the *cluster graph induced by* $\mathcal{P}$.

▶ **Lemma 2.** *For any positive integer* $k \geq 1$*, if the diameter of* $\overline{G}$ *satisfies* $\overline{D} \geq k\frac{\ln^2 n}{\beta^4}$*, then the diameter of the cluster graph* $\overline{G}^*$ *is at most* $2\beta\overline{D}$*, with probability at least* $1 - \frac{1}{n^{k-2}}$.

2.2 Rooted Spanning Tree

Let us now describe an asynchronous distributed algorithm to construct a low diameter rooted spanning tree, given a pre-specified root, in a time-efficient and (near) message-optimal manner – see Theorem 3. We assume that each node knows whether it is the pre-specified root prior to the start of the algorithm. We also assume initially that the diameter of the original graph, D, is known to the nodes. We explain how to remove this assumption at the end of the section.

▶ **Theorem 3.** *Given a graph* G *with* n *nodes,* m *edges and diameter* D*, as well as a distinguished node* R*, and a constant parameter* $1 \geq \varepsilon > 0$*, the asynchronous distributed Procedure **ST-Cons**(*ε*) computes an* $\tilde{O}(D^{1+\varepsilon})$*-diameter spanning tree rooted in* R *with termination detection, using* $\tilde{O}(D^{1+\varepsilon})$ *time with high probability and* $\tilde{O}(m)$ *messages with high probability.*

Brief Description. We construct the low diameter spanning tree in a two stage process. The first stage consists of building a sequence of increasingly coarser partitions of $G = (V, E)$. Each partition decomposes V into disjoint node sets, called clusters, with strong diameter $\tilde{O}(D^{1+\varepsilon})$; in fact, each cluster C is spanned by a tree $\hat{T}(C)$ of depth $\tilde{O}(D^{1+\varepsilon})$. (Unlike in Subsect. 2.1, this spanning tree is oriented away from the root.) The unique cluster containing the root node R will be denoted C_R. The cluster graph induced by the final partition (defined in Subsect. 2.1) has diameter $\tilde{O}(1)$. These partitions are obtained by simulating the synchronous MPX decomposition algorithm (see Subsect. 2.1) on G, then on the obtained cluster graph, and so on, for $i_m = \lceil \log_{1/(3\beta)} D \rceil$ times (where $\beta = \ln^{-\frac{1}{\varepsilon'}} n$ and $\varepsilon' \leq 1$ is to be derived in the analysis). In the second stage, we construct a breadth first search (BFS) tree T^{BFS} over the final cluster graph of phase 1, where the cluster C_R containing the pre-specified root R serves as the root of the BFS tree. We then use T^{BFS} to decide which edges of the original graph should be kept to obtain the desired rooted spanning tree $\tilde{T}$ of G with depth $\tilde{O}(D^{1+\varepsilon})$.

Detailed Description. Consider the initial graph $G(V, E) = G_0(V_0, E_0)$ and the initial trivial partition $\mathcal{P}_0$ in which each node $v \in V$ is its own cluster.

- **Stage 1:** The first stage consists of $i_m = \lceil \log_{1/(3\beta)} D \rceil$ phases, where $\beta = \ln^{-1/\varepsilon'} n$ and we assume $\ln \ln n \geq 2\varepsilon' \ln 3$. (If $\ln \ln n \leq 2\varepsilon' \ln 3 \leq 2 \ln 3$, then constructing a low diameter spanning tree efficiently is trivial.) Phase i starts with a partition $\mathcal{P}_{i-1}$ of V and the cluster graph induced by $\mathcal{P}_{i-1}$ is denoted by $G_{i-1}(V_{i-1}, E_{i-1})$. We simulate one instance of Procedure **MPX** (with parameter β) on G_{i-1} in an asynchronous setting by running an α-synchronizer between clusters, and within each cluster C, using the spanning tree $\hat{T}(C)$ to simulate the behavior of each cluster node of V_{i-1}. (Note that this well-known synchronizer is described in more detail in Appendix A.) More precisely, the root of the spanning tree $\hat{T}(C)$ simulate the behavior of cluster C (in the simulated Procedure **MPX**). To send a (same) message to its adjacent clusters, C broadcasts along $\hat{T}(C)$. To receive the message with the minimum ID (which is sufficient information for Procedure **MPX**), C convergecasts along $\hat{T}(C)$.

  The output is a partition $\mathcal{P}_i^*$ of V_{i-1} into disjoint (cluster node) sets $U_1, \ldots, U_t$ such that each U_j has a spanning tree T_j^{super} of depth $O(\frac{\ln n}{\beta})$. We transform $\mathcal{P}_i^*$ into a partition $\mathcal{P}_i$ of V, the node set of the *original* graph, into disjoint node sets $W_1, \ldots, W_t$, such that each W_j has a spanning tree $\hat{T}(W_j)$ of depth $O((\frac{\ln n}{\beta})^i)$. (In fact, we only show how to compute the spanning trees $\hat{T}(W_j)$, which induces the node sets W_j.)

  To transform $\mathcal{P}_i^*$ to $\mathcal{P}_i$, we use a simple Procedure **Transform**, sketched next. Recall that each cluster node in U_j keeps information about its parent in the spanning tree T_j^{super}. Procedure **Transform** consists of $2\frac{\ln n}{\beta}$ iterations. Each cluster node keeps an iteration counter and these counters are kept locally synchronized by running an α-synchronizer between cluster nodes. In the first iteration, the root cluster node C_R sends its ID to each adjacent cluster node C (which is its child in T_j^{super}) over the edges of the set $E_{inter} = \{(u, w) \in E(G) \mid u \in C_R, w \in C\}$, namely, all (original) inter-cluster edges between C_R and C. (Note that in fact, C_R sends its ID to all adjacent cluster nodes, but cluster nodes which are not children of C_R simply ignore that message.) Among these inter-cluster edges, every child cluster node C keeps $(u^*, w^*) = argmin_{(u,w)\in E_{inter}}\{id_w\}$, i.e., the edge whose endpoint w in C has the minimum ID. Cluster node C then reorients its tree $\hat{T}(C)$ to be rooted in w (and the inter-cluster edge is oriented towards w, i.e., from parent to child). In the next iteration, each C sends the ID of R to its children cluster nodes, if they exist, which in turn reorient their tree in the same fashion. After all iterations are done, the "combined" spanning tree $\hat{T}(W_j)$ is completed, and a simple broadcast allows all nodes in the newly computed cluster W_j to move on to the next phase. (Note that $\hat{T}(W_j)$ is oriented from the root outwards.)

- **Stage 2:** At the end of stage 1, the final partition decomposes V into clusters with strong diameter $\tilde{O}(D^{1+\varepsilon})$ and induces a cluster graph $G_f(V_f, E_f)$ of diameter $O(\log^{2+4/\varepsilon'} n)$; in fact, each cluster C is spanned by a tree $\hat{T}(C)$ of depth $\tilde{O}(D^{1+\varepsilon})$. During stage 2, the naive synchronous BFS tree construction algorithm (based on flooding, see [50]) is simulated on G_f for $O(\log^{2+4/\varepsilon'} n)$ rounds, where the designated root in V_f is the cluster C_R that contains the pre-specified root in V. Once again, this is done by running an α-synchronizer between clusters, and within each cluster, using the spanning tree $\hat{T}(C)$ to simulate the behavior of each cluster node C. After computing the BFS tree T^{BFS} on G_f, we use Procedure **Transform** – but this time for $O(\log^{2+4/\varepsilon'} n)$ rounds – to compute a spanning tree $\tilde{T}$ of G, similarly to stage 1. This final output $\tilde{T}$ is a $\tilde{O}(D^{1+\varepsilon})$ diameter spanning tree of G.

Analysis. Lemma 4 upper bounds, for each phase, the diameter of the cluster graph as well as that of the partition's clusters. Corollary 5 is obtained from Lemma 4 by considering the last phase. After which, we prove Theorem 3 using Lemma 4 and Corollary 5. The proofs of Lemma 4 and Corollary 5 are deferred to Appendix B.

▶ **Lemma 4.** *For each phase $1 \leq i \leq i_m$, (1) diam$(G_{i-1}) = \max\{(3\beta)^{i-1}D, O(\log^{2+4/\varepsilon'} n)\}$ w.h.p., and (2) each cluster C of the partition $\mathcal{P}_{i-1}$ is spanned (in the original graph G) by a tree $\hat{T}(C)$ with diam$(\hat{T}(C)) = (\frac{5 \ln n}{\beta})^{i-1}$.*

▶ **Corollary 5.** *At the end of phase i_m, (1) diam$(G_{i_m}) = O(\log^{2+4/\varepsilon'} n)$ w.h.p., and (2) each cluster C of the partition $\mathcal{P}_{i_m}$ is spanned (in the original graph G) by a tree $\hat{T}(C)$ with diam$(\hat{T}(C)) = \tilde{O}(D^{1+\varepsilon})$.*

Proof of Theorem 3. The correctness of the first stage follows from that of the simulation (using an α-synchronizer between clusters), Procedure **MPX** and Procedure **Transform**. Next, let us show the time and message complexity of the first stage. During each phase $1 \leq i \leq i_m$, Procedure **MPX** is simulated on G_{i-1} for $O(\frac{\log n}{\beta}) = \tilde{O}(1)$ rounds. Hence, each cluster C simulates $\tilde{O}(1)$ rounds. In each round, the cluster broadcasts once over the cluster's spanning tree $\hat{T}(C)$, sends one message per inter-cluster edge over to adjacent clusters, and convergecasts once over $\hat{T}(C)$. By Lemma 4, $\hat{T}(C)$ has depth $\tilde{O}(D^{1+\varepsilon})$. Hence, each round of Procedure **MPX** is simulated in at most $\tilde{O}(D^{1+\varepsilon})$ time and using $O(m)$. Adding up over all phases results in $\tilde{O}(D^{1+\varepsilon})$ time and $\tilde{O}(m)$ messages. Note that running an α-synchronizer (between the clusters) induces only an $\tilde{O}(1)$ message overhead per (inter-cluster) edge over all rounds, but no time overhead. Thus Procedure **MPX** is simulated in $\tilde{O}(D^{1+\varepsilon})$ time and using $\tilde{O}(m)$ messages. Similarly, in Procedure **Transform**, each cluster C simulates $\tilde{O}(1)$ rounds. In each round, the cluster broadcasts twice over the cluster's spanning tree $\hat{T}(C)$, sends one message per inter-cluster edge over to adjacent clusters, and convergecasts twice over $\hat{T}(C)$ (where the additional broadcast and convergecast allows to reorient $\hat{T}(C)$). Therefore, it can be seen that Procedure **Transform** also takes $\tilde{O}(D^{1+\varepsilon})$ time and uses $\tilde{O}(m)$ messages. Finally, the first stage has at most $i_m = \tilde{O}(1)$ phases, and thus takes $\tilde{O}(D^{1+\varepsilon})$ time and uses $\tilde{O}(m)$ messages.

By Corollary 5, the final cluster graph has a diameter of $O(\log^{2+4/\varepsilon'} n)$. Given that, the correctness of the second stage follows from that of the simulation (using an α-synchronizer between clusters), the naive synchronous BFS tree construction algorithm and Procedure **Transform**. As for the time and message complexity, the same approach (used for stage 1 above) shows that the second stage takes $\tilde{O}(D^{1+\varepsilon})$ time and uses $\tilde{O}(m)$ messages. ◀

Removing the Requirement of the Knowledge of D. In the previously described algorithm, we assumed that each node knew the value of D, the diameter of the original graph. This assumption can be removed by having each node guess the value of $D = 2^1, 2^2, \ldots$ until we arrive at the correct guess (an at most 2-approximation of D).

An issue that must be addressed, however, is that nodes need some way to determine whether they have correctly guessed the value of D or not. This can be done at the end of the second stage. Recall that the naive synchronous BFS tree construction is simulated for $O(\log^{2+4/\varepsilon'} n) = \tilde{O}(1)$ rounds. If the estimate of D is too small, the cluster graph obtained at the end of the first stage, G_f, may have diameter strictly greater than $O(\log^{2+4/\varepsilon'} n)$, in which case T^{BFS} may not cover the whole graph G_f. As a result, once $\tilde{T}$ is constructed from T^{BFS} using Procedure **Transform**, some nodes may exist outside the spanning tree $\tilde{T}$. This condition can be detected by the leaves of $\tilde{T}$ and a simple convergecast can be used to

check if this condition holds true. In case it does, the root of $\tilde{T}$ can initiate a broadcast over the entire original graph to update the guess of D and run the algorithm with this updated guess. (Note that if the estimate of D is too small, it may still happen that T^{BFS} covers the whole graph G_f, in which case we correctly compute a low diameter spanning tree $\tilde{T}$ of G and the algorithm terminates.)

This modification increases the time complexity of the algorithm by at most a constant factor, and its message complexity by a factor of at most $O(\log D)$.

3 The Asynchronous MST Algorithm

In this section, we develop a randomized algorithm to construct an MST with high probability for a given graph in $\tilde{O}(D^{1+\varepsilon} + \sqrt{n})$ time with high probability and $\tilde{O}(m)$ messages with high probability (for any constant $\varepsilon > 0$).

3.1 High-level Overview of the Algorithm

We implement on an asynchronous network a variant of the singularly near optimal *synchronous* MST algorithms of [18, 47]. The algorithm can be divided into three stages. In stage I, we pre-process the network so that subsequent processes are fast and message efficient. Stages II and III correspond to the actual MST algorithm.

In order to ensure that nodes participate in this multi-stage algorithm in the proper sequence, we append a constant number of bits to each message to indicate the stage number that message corresponds to. A node u knows which stage number it is currently in and can queue received messages that belong to a later stage. These messages will be processed later, once u reaches to the corresponding stage.

Stage I: Pre-Processing the Graph. In this stage, we run a few preparatory procedures on the graph. Specifically, we first elect a leader, then construct a low diameter spanning tree $\mathcal{T}$, and finally estimate the diameter of $\mathcal{T}$. In more detail, for the first stage we utilize the singularly (near) optimal algorithm of [37] to elect a unique leader $\mathcal{L}$ in $O(D + \log^2 n)$ time and $O(m \log^2 n)$ messages. Subsequently, we run the **ST-Cons**(ε) algorithm of Section 2 (for a constant parameter $1 \geq \varepsilon > 0$) to construct a low diameter spanning tree $\mathcal{T}$ on G rooted at $\mathcal{L}$. Then, we use a known application of the Wave&Echo technique (see, e.g., [54, 58]) to have the root calculate the diameter of the constructed spanning tree D', which we know is an $\tilde{O}(D^{\varepsilon})$ approximation of the diameter D of the original graph G, in $O(D')$ time and $O(n)$ messages. Finally, all nodes in the tree participate in a simple broadcast on the spanning tree $\mathcal{T}$ to send this knowledge of D' to all nodes in the graph in $O(D')$ time and $O(n)$ messages.

Stage II: Controlled-GHS. The **Controlled-GHS** algorithm, introduced in [23, 39], is a *synchronous* version of the classical Gallager-Humblet-Spira (GHS) algorithm [22, 50] with some modifications, aiming to balance the size and diameter of the resulting fragments. Here, we convert to the asynchronous setting a variant of the (synchronous) **Controlled-GHS** as described in [47, 49].

Recall that the synchronous GHS algorithm (see, e.g., [50]) consists of $O(\log n)$ phases. In the initial phase, each node is an *MST fragment*, by which we mean a connected subgraph of the MST. In each subsequent phase, every MST fragment finds a minimum-weight outgoing edge (MOE) – these edges are guaranteed to be in the MST [57]. The MST fragments are merged via the MOEs to form larger fragments. The number of phases is $O(\log n)$, since the number of MST fragments gets at least halved in each phase. The message

complexity is $O(m + n \log n)$, which is essentially optimal, and the time complexity is $O(n \log n)$. Unfortunately, the time complexity of the GHS algorithm is not optimal, because much of the communication during a phase uses *only the MST fragment edges*, and the diameter of an MST fragment can be significantly larger than the graph diameter D (possibly as large as $\Omega(n)$).

In order to obtain a time-optimal algorithm, the **Controlled-GHS** algorithm controls the growth of the diameter of the MST fragments during merging. This is achieved by computing, in each phase, a maximal matching on the fragment forest with additional edges being carefully chosen to ensure enough fragments merge together, and merging fragments accordingly. Each phase essentially reduces the number of fragments by a factor of two, while not increasing the diameter of any fragment by more than a factor of two. Since the number of phases of **Controlled-GHS** is capped at $\max\{\lceil \log_2 \sqrt{n} \rceil, \lceil \log_2 D' \rceil\}$, it produces at most $\min\{\sqrt{n}, n/D'\}$ fragments, each of which has diameter $O(D' + \sqrt{n})$. These are called *base fragments.* **Controlled-GHS** up to phase $\max\{\lceil \log_2 \sqrt{n} \rceil, \lceil \log_2 D' \rceil\}$ can be implemented using $\tilde{O}(m)$ messages in $\tilde{O}(D' + \sqrt{n})$ rounds in a synchronous network.

Stage II executes the **Controlled-GHS** algorithm in an asynchronous network. We postpone the discussion of the technical details involved in efficiently implementing the asynchronous algorithm to Section 3.2. The main challenge, however, is that the synchronous version heavily relies on the phases being synchronized. Here, we cannot naively use a synchronizer (such as α) for synchronization, as it would have increased the message complexity substantially. Instead we use a light-weight synchronization that incurs only $\tilde{O}(m)$ overhead in messages.

Finally, we ensure that all nodes know the exact number of fragments that were constructed at the end of this phase. The root of each fragment T calculates the number of nodes present in T and forms a tuple consisting of this value and the ID of T. Subsequently, each fragment root participates in the upcast of its tuple in the low diameter spanning tree $\mathcal{T}$ on G'. All tuples are accumulated at $\mathcal{L}$ in $O(\min\{\sqrt{n}, n/D'\} + D')$ time and $O(n)$ messages. $\mathcal{L}$ continues to listen for messages until the total number of nodes in all fragments it has heard from is equal to n, i.e., all fragments have been heard from. Now $\mathcal{L}$ broadcasts the number of fragments over $\mathcal{T}$ to all nodes in the graph in $O(D')$ time and $O(n)$ messages.

Stage III: Merging the Remaining Fragments. This stage completes the fragment merging process. However, the merging is done in a "soft" manner. The at most $\min\{\sqrt{n}, n/D'\}$ base fragments (constructed at the end of Stage II) are still retained, but each base fragments takes on an additional ID–a cluster ID, initially set to the base fragment ID. (A cluster is a collection of base fragments; at the beginning of this stage, each base fragment forms its own cluster.) Each base fragment finds an MOE to a different cluster, if such an MOE exists, and merging consists of base fragments modifying their associated cluster IDs and marking the corresponding MOE connecting clusters. All nodes participate in a simple upcast over $\mathcal{T}$, where the root of each base fragment is responsible to send up a tuple consisting of its fragment & cluster IDs, a possible MOE and the associated fragment & cluster IDs the MOE leads to.[7] It is similar to the approach of [18, 47], which uses a BFS tree to upcast these values to the root of tree; here, instead of BFS, we use the low-diameter spanning tree of

[7] It is required that each base fragment's root sends up this tuple even if it does not have an MOE (in which case the tuple only has info on the fragment ID and cluster ID of the base fragment). This is to ensure that the nodes detect termination as the root of $\mathcal{T}$, $\mathcal{L}$, already knows the fragment and cluster IDs of the base fragments so it knows how many such messages to wait for.

Section 2. Subsequently, the root calculates the appropriate MOEs (and the fragments they connect and the clusters they lead to) for each cluster and downcast these values. Each fragment then performs a broadcast of its (possibly new) cluster ID over the fragment tree (to all nodes within the fragment). This process is repeated for $O(\log n)$ phases until only one cluster remains, which represents the MST of the original graph.

Let us examine each phase i in more detail. Each base fragment finds its respective MOE, if any, and sends it to $\mathcal{L}$ via an upcast.[8] All fragment leaders can find their MOEs in $O(D' + \sqrt{n})$ time and $O(m)$ messages. Upcasting these values to $\mathcal{L}$ using tree $\mathcal{T}$ takes $O(\min\{\sqrt{n}, n/D'\} + D')$ time and $O(n)$ messages. $\mathcal{L}$ locally computes the overall MOEs of the (soft-merged) base fragments and then merges them (locally). Subsequently, all nodes of $\mathcal{T}$ participate in a downcast of these MOEs and modified cluster IDs (that $\mathcal{L}$ previously calculated) in $O(D' + \sqrt{n})$ time and $O(n)$ messages. Each base fragment performs a broadcast of its (possibly new) cluster ID to all nodes in its base fragment utilizing the base fragment tree. For all base fragments to do this, it takes a total of $O(D' + \sqrt{n})$ time and $O(n)$ messages.

3.2 Detailed Algorithm Description

We now look at each stage in more detail.

Stage I. In this stage, the nodes first run Procedure **LE** on G to elect a unique leader $\mathcal{L}$ with high probability. As a side benefit, the procedure also wakes up all nodes. Next, the nodes participate in Procedure **ST-Cons**(ε) to construct an $\tilde{O}(D^{1+\varepsilon})$ diameter spanning tree $\mathcal{T}$ of G with $\mathcal{L}$ as its root. Subsequently, all nodes participate in Procedure **Diam-Calc** so that $\mathcal{L}$ is now aware of the diameter D' of $\mathcal{T}$. Finally, all nodes participate in **Frag-Bcast** over $\mathcal{T}$ to transmit this information of D' to all nodes in the graph. (Procedures **LE**, **Diam-Calc** and **Frag-Bcast** are described in Appendix A.)

Stage II. In this stage, the nodes execute an asynchronous version of the **Controlled-GHS** algorithm [23, 47, 49]. Let us first recall the original (synchronous) **Controlled-GHS** algorithm. This algorithm merges fragments (subtrees of the MST) in phases, similarly to GHS. However, it guarantees two additional properties to hold at the end of each phase i: (a) there are at most $n/2^i$ fragments, and (b) each fragment has diameter $O(2^i)$. These guarantees are ensured through two measures. First, at the beginning of phase i, only fragments with diameter $\leq 2^i$ will participate in this phase and find MOEs. Second, in a phase i, consider the *fragment graph* whose "nodes" are the fragments (including those that do not participate) and whose edges are all the MOEs found. The algorithm first performs a maximal matching on this fragment graph and removes from the fragment graph edges that do not participate in this matching. Then, those fragments who participate in this phase and remain unmatched add their MOEs back to the fragment graph. Connected components of fragments in this final fragment graph then merge together. The algorithm is run from phase $i = 0$ to phase $i = \max\{\lceil \log_2 \sqrt{n} \rceil, \lceil \log_2 D' \rceil\}$. Due to a lack of space, the details of the adaptation of the **Controlled-GHS** algorithm to the asynchronous setting can be found in the full version of the paper.

[8] Note that as the algorithm progresses, two adjacent base fragments may belong to the same overall cluster, possibly resulting in one of those base fragments having no MOE to a different cluster.

After completing the last phase of the above process, we are almost ready to move to stage III of the algorithm.[9] Some final cleanup is first needed. We need two things in order to ensure our subsequent upcasts and downcasts over $\mathcal{T}$ have termination detection: (i) $\mathcal{L}$ needs to be made aware of how many base fragments are present and their IDs and (ii) each node in $\mathcal{T}$ needs routing information related to any fragment roots located in the subtree rooted at that node in $\mathcal{T}$.[10]

We need each fragment F to inform $\mathcal{L}$ of its existence and fragment ID. Now, the root of each fragment F, with ID ID_F, initiates **Tree-Count** to determine the number of nodes in the fragment, $size_F$. (Procedure **Tree-Count** is described in Appendix A.) Subsequently, all nodes in the graph participate in Procedure **Upcast** over $\mathcal{T}$ where each base fragment's root sends up the tuple $\langle ID_F, size_F\rangle$.[11] $\mathcal{L}$ accumulates these messages until $\sum_F size_F = n$, at which point $\mathcal{L}$ knows the exact number of base fragments, say NUM-OF-BASE-FRAGMENTS, and their IDs. Once $\mathcal{L}$ recognizes that it has received all the messages, it initiates a broadcast of NUM-OF-BASE-FRAGMENTS over $\mathcal{T}$. Now all nodes are aware of the number of base fragments.

Stage III. In this stage, each node u maintains two sets of variables. One set of variables relates to the base fragment B node u it belongs to at the end of phase two. These variables store information about the base fragment such as the base fragment ID ID_B, u's parents in B, and u's children in B. The second set of variables relates to what we term a *cluster*, a connected subgraph in $\mathcal{H}$ consisting of base fragments and MOEs between them, and they store information that includes a cluster ID and cluster edges. Each node belonging to base fragment B initially sets its cluster ID $\mathsf{CLUSTER\text{-}ID}_B$ to be the same as its base fragment ID. Each node u also stores a set of cluster edges adjacent to it in the set $\mathsf{CLUSTER\text{-}EDGES}_u$, which is initially empty. Edges are added to $\mathsf{CLUSTER\text{-}EDGES}_u$ in the course of stage III. At the end of stage III, for a given node u, the set of edges in the MST is the union of the set of edges in $\mathsf{CLUSTER\text{-}EDGES}_u$ and its children and parent in B. Node $\mathcal{L}$ maintains, in addition, information on the supergraph $\mathcal{H}$ formed by the base fragments (including the updated cluster IDs of those base fragments) and any MOE edges that $\mathcal{L}$ computes in the phases of stage III, to be described below.

In stage III, each node participates in the following process for $\lceil \log_2 n \rceil$ phases until it terminates. Once again, nodes use a β-synchronizer over $\mathcal{T}$ to keep track of the phase number in stage III. In each phase, each base fragment B with root R_B, fragment ID ID_B, and cluster ID $\mathsf{CLUSTER\text{-}ID}_B$ runs Procedure **Find-MOE** to find its minimum outgoing edge, say $\mathsf{MOE\text{-}VALUE}_B$, to a node with a different cluster ID, if there is any. All nodes in the graph then participate in Procedure **Upcast** over $\mathcal{T}$ to send informatino on the fragments up to $\mathcal{L}$. Specifically, each base fragment B's root sends up the tuple consisting of information on B as well as the computed MOE, if any.

Once $\mathcal{L}$ receives this tuple from all base fragments, it locally computes the MOE edges for each cluster in the supergraph $\mathcal{H}$. Recall that a cluster is a connected subgraph of base fragments in $\mathcal{H}$. Thus, the MOE from a cluster is really an MOE from one of the base

[9] As we use a β-synchronizer to keep track of which phase a node is in, it is possible to know when $\max\{\lceil \log_2 \sqrt{n} \rceil, \lceil \log_2 D' \rceil\}$ phases are over.

[10] Consider a node u and let node v be the root of a fragment located in the subtree rooted at u in $\mathcal{T}$. We say node u has routing information on v when u knows which of its children in $\mathcal{T}$ to send a message destined for v

[11] It is important to note that during Procedure **Upcast**, each node u in $\mathcal{T}$ learns about which of its children in $\mathcal{T}$ lead to which fragment roots. In other words, u learns routing information related to any fragments roots located in the subtree in $\mathcal{T}$ rooted at u, satisfying our second requirement from the previous paragraph.

fragments that constitutes it. Define $\mathsf{FINAL\text{-}MOE\text{-}VALUE}_B$ as the MOE, if any, for base fragment B. For each base fragment B, $\mathcal{L}$ computes its new cluster ID $\mathsf{CLUSTER\text{-}ID}_B$ (if multiple clusters merge, the smallest cluster ID becomes the ID of the new merged cluster), and its $\mathsf{FINAL\text{-}MOE\text{-}VALUE}_B$ (if the original value of $\mathsf{FINAL\text{-}MOE\text{-}VALUE}_B$ broadcast by B was selected as a new edge in $\mathcal{H}$, $\mathsf{FINAL\text{-}MOE\text{-}VALUE}_B$ is set to $\mathsf{MOE\text{-}VALUE}_B$, else it is set to a null value).

All nodes participate in Procedure **Downcast** so that $\mathcal{L}$ may inform each base fragment's root about its possibly new cluster ID and MOE edge. (Procedure **Downcast** is described in Appendix A.) Subsequently each base fragment participates in Procedure **Frag-Bcast** to send these values to all nodes in the fragment. Each node updates its cluster ID if needed. If there is information on a new MOE edge out of one of the nodes u, then u adds this edge to $\mathsf{CLUSTER\text{-}EDGES}_u$. Once the final phase of stage III is complete, all nodes terminate the algorithm.

4 Analysis of the MST Algorithm

We argue that Algorithm **Sing-MST** correctly outputs the MST with high probability and subsequently analyze its running time and message complexity.

It is easy to see that the algorithm faithfully simulates **Controlled-GHS** in the asynchronous setting. Recall that **Controlled-GHS** requires us to maintain two properties in each phase of the algorithm: (i) at the end of phase i, there are at most $n/2^i$ fragments and (ii) at the end of phase i, each fragment has diameter $O(2^i)$. Since the algorithm faithfully simulates **Controlled-GHS**, it follows from the analysis of **Controlled-GHS** (see e.g., [18, 47]) that these properties are maintained in stage II. In stage III, they are also maintained via the "soft merge" process in a way that is time and message efficient. Note that in stage III, we ensure that those properties hold now on clusters instead of on fragments. These two properties guarantee that after the algorithm is over, there exists one cluster such that all nodes belong to the cluster and the only edges in the cluster are MST edges of the original graph. The high probability guarantee comes from the usage of (randomized) Procedures **LE** and **ST-Cons**.

We now bound the running time and message complexity in each stage of the algorithm. Due to a lack of space, the proofs of the following lemmas are deferred to the full version.

▶ **Lemma 6.** *Stage I of Algorithm **Sing-MST** takes $\tilde{O}(D^{1+\varepsilon})$ time with high probability and $\tilde{O}(m)$ messages with high probability, for any constant $\varepsilon > 0$.*

▶ **Lemma 7.** *Stage II takes $\tilde{O}(D^{1+\varepsilon} + \sqrt{n})$ time and $\tilde{O}(m)$ messages.*

▶ **Lemma 8.** *Stage III takes $\tilde{O}(D^{1+\varepsilon} + \sqrt{n})$ time and $\tilde{O}(m)$ messages to complete.*

By Lemmas 6, 7, and 8 and our initial discussion about correctness, we get the following theorem.

▶ **Theorem 9.** *Algorithm **Sing-MST** computes the minimum spanning tree of an arbitrary graph with high probability in the asynchronous KT_0 $\mathcal{CONGEST}$ model in $\tilde{O}(D^{1+\varepsilon} + \sqrt{n})$ time with high probability and $\tilde{O}(m)$ messages with high probability. Furthermore, nodes know their edges in the MST and terminate when the algorithm is over.*

As a consequence of the above theorem and a theorem due to Mashregi and King [44, Theorem 1.2] we also get the following result in the KT_1 model.

▶ **Theorem 10.** *There is an asynchronous algorithm that computes the minimum spanning tree of an arbitrary graph with high probability in the asynchronous KT_1 $\mathcal{CONGEST}$ model in $\tilde{O}(D^{1+\varepsilon} + n^{1-2\delta})$ time and $\tilde{O}(n^{3/2+\delta})$ messages for any small constant $\varepsilon > 0$ and for any $\delta \in [0, 0.25]$.*

The above theorem gives the first asynchronous MST algorithm in the KT_1 $\mathcal{CONGEST}$ model that has *sublinear time* (for all $D = O(n^{1-\varepsilon'})$ for any arbitrarily small constant $\varepsilon' > 0$) and *sublinear* messages complexity.

5 Conclusion and Open Problems

Recall that while most of the paper deals with the common KT_0 model, Theorem 10 includes a contribution also under the KT_1 model. This model has grown in popularity in recent years first because one can claim it is a more natural model [8] and second because it allows reducing the communication to $o(m)$. Initially, it looked as if this reduction carries a significant cost in time complexity, trading off the attempt to go below $\Omega(n)$ when the diameter is smaller [33]. This went against the direction for the KT_0 model, where algorithms managed to be efficient both in time complexity and message complexity [18, 47, 28, 26]. Those results, however, were in the synchronous model. Theorem 10 (together with [44][Theorem 1.2]) is the first result that approaches optimal time while keeping message complexity $o(m)$. It would be interesting to see whether this is the best that can be obtained in this direction. Results showing that other tasks can be obtained with $o(m)$ messages but time efficiently in KT_1 would also be interesting.

The asynchronous distributed MST algorithm for KT_0 presented here continues a long line of work in distributed MST algorithms. Our algorithm essentially (up to a polylog(n) factor) matches the respective time and message lower bounds, but for an arbitrarily small constant factor ε in the exponent of D (with respect to time). Yet, several open problems remain. Is it possible to achieve near singular optimality? That is, can we achieve optimality within a polylog(n) factor in both time and messages? This seems related to constructing a $\tilde{O}(D)$ diameter spanning tree in a singularly optimal fashion which is also open. Our low-diameter spanning tree construction comes close to achieving this, but for a $\tilde{O}(D^{\varepsilon})$ factor in the diameter and run time. This is also closely related to constructing a BFS (or nearly BFS) tree in a singularly optimal fashion.

The tools and techniques used in this paper for accomplishing various tasks in a (almost) singularly optimal fashion in an asynchronous setting can also be useful in solving other fundamental problems such as shortest paths, minimum cut etc. In particular, the techniques of this paper can be useful in showing that the partwise aggregation operation of Ghaffari and Haeupler [25] can be implemented in the asynchronous setting in $\tilde{O}(D^{1+\varepsilon} + \sqrt{n})$ and $\tilde{O}(m)$ messages. This would imply that problems such as exact minimum cut and $(1+\varepsilon)$-single source shortest path can be solved almost singularly optimally. We will elaborate on these in more detail in the full version of the paper.

For our singularly optimal algorithms we focused on being (existentially) optimal in time with respect to parameters n and D (i.e., with respect to the $\tilde{\Omega}(D + \sqrt{n})$ bound). An interesting direction of future work is obtaining asynchronous algorithms that are "universally optimal" (Haeupler, Wajc, and Zuzic [32]) (with respect to time) and also optimal with respect to messages.

References

1. Yehuda Afek and Eli Gafni. Time and message bounds for election in synchronous and asynchronous complete networks. *SICOMP*, 20(2):376–394, 1991.
2. Yehuda Afek and Yossi Matias. Elections in anonymous networks. *Information and Computation*, 113(2):312–330, 1994.
3. John Augustine, Seth Gilbert, Fabian Kuhn, Peter Robinson, and Suman Sourav. Latency, capacity, and distributed minimum spanning tree. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 157–167. IEEE, 2020.
4. Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, 1985.
5. Baruch Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *Proceedings of the 19th ACM Symposium on Theory of Computing (STOC)*, pages 230–240, 1987.
6. Baruch Awerbuch. Distributed shortest paths algorithms (extended abstract). In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 490–500, 1989.
7. Baruch Awerbuch, Oded Goldreich, Ronen Vainish, and David Peleg. A trade-off between information and communication in broadcast protocols. *J. ACM*, 37:238–256, 1990.
8. Baruch Awerbuch, Oded Goldreich, Ronen Vainish, and David Peleg. A trade-off between information and communication in broadcast protocols. *Journal of the ACM (JACM)*, 37(2):238–256, 1990.
9. Baruch Awerbuch and David Peleg. Network synchronization with polylogarithmic overhead. In *31st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 514–522, 1990.
10. Yi-Jun Chang, Varsha Dani, Thomas P. Hayes, Qizheng He, Wenzheng Li, and Seth Pettie. The energy complexity of broadcast. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, pages 95–104, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3212734.3212774.
11. Yi-Jun Chang, Varsha Dani, Thomas P. Hayes, and Seth Pettie. The energy complexity of bfs in radio networks. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, pages 273–282, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3382734.3405713.
12. Yogen K Dalal. *A Distributed Algorithm for Constructing Minimal Spanning Trees in Computer-Communication Networks*. Stanford University, 1976.
13. Yogen K. Dalal. A distributed algorithm for constructing minimal spanning trees. *IEEE Trans. Software Eng.*, 13(3):398–405, 1987.
14. Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. *SIAM J. Comput.*, 41(5):1235–1265, 2012.
15. Devdatt P. Dubhashi and Alessandro Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, 2009. URL: http://www.cambridge.org/gb/knowledge/isbn/item2327542/.
16. Michael Elkin. A faster distributed protocol for constructing minimum spanning tree. *Journal of Computer and System Sciences*, 72(8):1282–1308, 2006.
17. Michael Elkin. An unconditional lower bound on the time-approximation trade-off for the distributed minimum spanning tree problem. *SIAM J. Comput.*, 36(2):433–456, 2006.
18. Michael Elkin. A simple deterministic distributed MST algorithm, with near-optimal time and message complexities. In *Proceedings of the 2017 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 157–163, 2017.
19. Michael Elkin, Hartmut Klauck, Danupon Nanongkai, and Gopal Pandurangan. Can quantum communication speed up distributed computation? In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 166–175. ACM, 2014.

20 Michalis Faloutsos and Mart Molle. A linear-time optimal-message distributed algorithm for minimum spanning trees. *Distributed Computing*, 17(2):151–170, 2004.

21 Pierre Fraigniaud, Amos Korman, and Emmanuelle Lebhar. Local mst computation with short advice. *Theory of Computing Systems*, 47(4):920–933, 2010.

22 Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, 1983.

23 Juan A. Garay, Shay Kutten, and David Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM J. Comput.*, 27(1):302–316, 1998.

24 Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks II: low-congestion shortcuts, mst, and min-cut. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 202–219. SIAM, 2016.

25 Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks II: low-congestion shortcuts, MST, and min-cut. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 202–219, 2016.

26 Mohsen Ghaffari and Fabian Kuhn. Distributed MST and broadcast with fewer messages, and faster gossiping. In *Proceedings of the 32nd International Symposium on Distributed Computing (DISC)*, pages 30:1–30:12, 2018.

27 Robert Gmyr and Gopal Pandurangan. Time-message trade-offs in distributed algorithms. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, pages 32:1–32:18, 2018.

28 Robert Gmyr and Gopal Pandurangan. Time-message trade-offs in distributed algorithms. In *Proceedings of the 32nd International Symposium on Distributed Computing (DISC)*, pages 32:1–32:18, 2018.

29 Sandeep KS Gupta and Pradip K Srimani. Self-stabilizing multicast protocols for ad hoc networks. *Journal of Parallel and Distributed Computing*, 63(1):87–96, 2003.

30 Bernhard Haeupler, D. Ellis Hershkowitz, and David Wajc. Round-and message-optimal distributed graph algorithms. In *PODC*, pages 119–128, 2018.

31 Bernhard Haeupler and David Wajc. A faster distributed radio broadcast primitive: Extended abstract. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 361–370, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2933057.2933121.

32 Bernhard Haeupler, David Wajc, and Goran Zuzic. Universally-optimal distributed algorithms for known topologies. In *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 1166–1179. ACM, 2021.

33 Valerie King, Shay Kutten, and Mikkel Thorup. Construction and impromptu repair of an MST in a distributed network with $o(m)$ communication. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 71–80, 2015.

34 Liah Kor, Amos Korman, and David Peleg. Tight bounds for distributed MST verification. In *Proc. 28th Symp. on Theoretical Aspects of Computer Science (STACS)*, volume 9 of *LIPIcs*, pages 69–80. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011.

35 Amos Korman and Shay Kutten. Distributed verification of minimum spanning trees. *Distributed Computing*, 20(4):253–266, 2007.

36 Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. In *Proc.24th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 9–18, 2005.

37 Shay Kutten, William K. Moses Jr., Gopal Pandurangan, and David Peleg. Singularly near optimal leader election in asynchronous networks. In *35th International Symposium on Distributed Computing (DISC)*, pages 27:1–27:18, 2021.

38 Shay Kutten, Gopal Pandurangan, David Peleg, Peter Robinson, and Amitabh Trehan. On the complexity of universal leader election. *J. ACM*, 62(1), 2015.

39 Shay Kutten and David Peleg. Fast distributed construction of small k-dominating sets and applications. *J. Algorithms*, 28(1):40–66, 1998.

40 Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-weight spanning tree construction in $O(\log\log n)$ communication rounds. *SIAM J. Comput.*, 35:120–131, 2005.

41 Zvi Lotker, Boaz Patt-Shamir, and David Peleg. Distributed MST for constant diameter graphs. In *Proc. 20th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 63–71, 2001.

42 Ali Mashreghi and Valerie King. Time-communication trade-offs for minimum spanning tree construction. In *Proceedings of the 18th International Conference on Distributed Computing and Networking (ICDCN)*, 2017.

43 Ali Mashreghi and Valerie King. Broadcast and minimum spanning tree with o(m) messages in the asynchronous CONGEST model. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, volume 121 of *LIPIcs*, pages 37:1–37:17, 2018.

44 Ali Mashreghi and Valerie King. Brief announcement: Faster asynchronous MST and low diameter tree construction with sublinear communication. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, volume 146 of *LIPIcs*, pages 49:1–49:3, 2019.

45 Ali Mashreghi and Valerie King. Broadcast and minimum spanning tree with o(m) messages in the asynchronous CONGEST model. *Distributed Computing*, pages 1–17, 2021.

46 Gary L. Miller, Richard Peng, and Shen Chen Xu. Parallel graph decompositions using random shifts. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 196–203, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2486159.2486180.

47 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. A time- and message-optimal distributed algorithm for minimum spanning trees. In *Proceedings of the 49th Annual ACM Symposium on the Theory of Computing (STOC)*, pages 743–756, 2017.

48 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. The distributed minimum spanning tree problem. *Bulletin of the EATCS*, 125, 2018.

49 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. A time- and message-optimal distributed algorithm for minimum spanning trees. *ACM Transactions on Algorithms (TALG)*, 16(1):1–27, 2019.

50 David Peleg. *Distributed Computing: A Locality Sensitive Approach.* SIAM, 2000.

51 David Peleg and Vitaly Rubinovich. A near-tight lower bound on the time complexity of distributed minimum-weight spanning tree construction. *SIAM J. Comput.*, 30(5):1427–1442, 2000.

52 Deepak Rohilla, Mahendra Kumar Murmu, and Shashidhar Kulkarni. An efficient distributed approach to construct a minimum spanning tree in cognitive radio network. In *First International Conference on Sustainable Technologies for Computational Intelligence*, pages 397–407. Springer, 2020.

53 Baruch Schieber and Marc Snir. Calling names on nameless networks. *Information and Computation*, 113(1):80–101, 1994.

54 Adrian Segall. Distributed network protocols. *IEEE transactions on Information Theory*, 29(1):23–35, 1983.

55 Gurdip Singh. Efficient leader election using sense of direction. *Distributed Computing*, 10(3):159–165, 1997. doi:10.1007/s004460050033.

56 Philip Spira. Communication complexity of distributed minimum spanning tree algorithms. In *Proceedings of the second Berkeley conference on distributed data management and computer networks*, 1977.

57 Robert Endre Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 1983.

58 Gerard Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.

A Toolbox

In this section, we present several procedures that are used as blackboxes in the current paper. As these procedures are either from other papers or minor variations of those in other papers, we merely mention what they do and their guarantees here.

Synchronization

Synchronizers are mechanisms that allow nodes to run synchronous algorithms in an asynchronous network with some overhead, either in time or messages.

α-synchronizer. An *alpha*-synchronizer, presented by Awerbuch [4], is a well known mechanism for nodes to run synchronous algorithms in an asynchronous network in the same running time (with a diameter overhead to time) while suffering a message overhead equivalent to the product of the run time of the synchronous algorithm and $O(m)$. Informally, when simulating some synchronous algorithm Alg, each node v sends a "pulse" message to all its neighbors after all of v's messages in the current round of Alg were acknowledged. Thus, v's neighbors can keep track of which pulse, or "clock tick", v has simulated. Additionally, note that it takes $O(D)$ time to initialize the α-synchronizer. A good description appears also in [50]. We know the following about an α-synchronizer.

▶ **Lemma 11** (Adapted from [50]). *Consider a graph G with n nodes, m edges, and diameter D in an asynchronous setting. The nodes of the graph may simulate a synchronous algorithm that takes $O(T)$ rounds and $O(M)$ messages in the synchronous setting by utilizing an α-synchronizer. The resulting simulated algorithm takes $O(T + D)$ time and $O(M + Tm)$ messages and has termination detection.*

β-synchronizer. A β-synchronizer is another type of synchronizer that reduces the message overhead at the expense of time. An assumption is made that there exists a spanning tree $\mathcal{T}$, rooted at some node $\mathcal{L}$, of depth d overlaid on top of the original graph and that each node knows its parent and children in the tree, if any. Now, as with the α-synchronizer, a synchronous algorithm that takes $O(T)$ rounds and $O(M)$ messages may be simulated in an asychronous network with the help of pulses. However, here each node sends a pulse to its parent once the current round is done and it has received pulses from each of its children in the tree. Once the root receives the pulse and finishes the current round, it broadcasts a message to move to the next round along the tree. The resulting simulated algorithm takes $O(T \cdot d)$ time and $O(M + Tn)$ messages.

▶ **Lemma 12** (Adapted from [50]). *Consider a graph G with n nodes in an asynchronous setting. Assume that there exists a rooted spanning tree $\mathcal{T}$ of depth d overlaid on G such that each node knows its parent and children, if any, in the tree. The nodes of the graph may simulate a synchronous algorithm that takes $O(T)$ rounds and $O(M)$ messages in the synchronous setting by utilizing a β-synchronizer over $\mathcal{T}$. The resulting simulated algorithm takes $O(T \cdot d)$ time and $O(M + Tn)$ messages and has termination detection.*

Notice that both α- and β-synchronizers can be used by nodes to enact a type of global round counter up to any number that can be encoded using $O(\log n)$ bits.

Leader Election

We make use of the leader election procedure, call it Procedure **LE**, of Kutten et al. [37] to elect a leader with high probability. Adapting Theorem 11 to this setting, we have the following lemma. Note that in the course of the procedure, all nodes are woken up but such information was not mentioned in the theorem statement in [37], so we add it here.

▶ **Lemma 13** (Theorem 11 in [37]). *Procedure **LE** solves leader election with termination detection with high probability in any arbitrary graph with n nodes, m edges, and diameter D in $O(D + \log^2 n)$ time with high probability using $O(m \log^2 n)$ messages with high probability in an asynchronous system with adversarial node wake-up. At the end of the procedure, all nodes are awake.*

Operations on a Fragment

In the course of our algorithm, we reach a situation where the graph G is partitioned into a set of disjoint trees (called fragments), each with a distinct root, an associated fragment ID, and an associated cluster ID (which may be different from its fragment ID). Each node knows its parent and children in the fragment, if any. We now describe some common operations that are to be performed on such trees.

Consider a tree T spanning a subset of the nodes of G, oriented towards a distinct root R. Let the tree have fragment ID F, known to all nodes in T. Furthermore, all nodes of T have the same cluster ID, say C, which may or may not be equal to F. Let $size(T)$ and $depth(T)$ denote the number of vertices and the depth of T, respectively.

Broadcast on a Fragment. Suppose a message M, originating at the root R, must be distributed to all nodes of the tree. Procedure **Frag-Bcast** performs this operation in a straightforward manner. The root R sends M to all its neighbors. Intermediate nodes receiving M on some round forward it to all their children in T in the next round.

To ensure termination detection, the procedure then performs a *convergecast* of acknowledgements on T as follows. Each leaf, upon receiving M, sends back an "ack" message. Each intermediate node waits until it receives an "ack" from all its children, and then sends an "ack" to its parent. The operation terminates once the root receives an "ack" from all its children.

▶ **Lemma 14.** *Procedure **Frag-Bcast**, run by nodes in the tree T, performs broadcast of a message originating at the root of T with termination detection in $O(depth(T))$ time and $O(size(T))$ messages.*

Upcast on a Fragment. Suppose m distinct and uncombinable messages, originating at arbitrary locations in the tree, must be gathered to the root R. Procedure **Upcast** performs this operation in a straightforward manner. Each node in the tree pipelines the messages it has seen upwards in the tree (towards R), in some arbitrary order.

We assume that R knows the number m of such messages it expects to receive and ensure this is true everywhere the procedure is called. Thus, R knows when it has received all m messages. To ensure termination detection, the procedure then performs **Frag-Bcast**.

▶ **Lemma 15.** *Procedure **Upcast**, run by nodes in the tree T, performs upcasting of m distinct messages with termination detection in $O(m + depth(T))$ time and $O(m \cdot depth(T))$ messages.*

Downcast on a Fragment. Suppose m distinct and uncombinable messages $M_1, \ldots, M_m$, originating at the root R, must be distributed to arbitrary destinations $w_1, \ldots, w_m$ in the tree, respectively. Procedure **Downcast** performs this operation in a straightforward manner. In each round i, R sends the pair (M_i, w_i) to its neighbor on the unique R-w_i path in T. Intermediate nodes receiving a pair (M_i, w_i) on some round forward it towards w_i in the next round. (Note that tie-breaking is not required.)

To ensure termination detection, the procedure then performs a convergecast of acknowledgements, backtracking on the subtree T' marked by the downcast messages; namely, each intermediate node that received ℓ messages from its parent and forwarded ℓ_j messages to its child x_j expects "ack - ℓ_j" from x_j. After receiving all such "ack" messages from its children, it sends "ack - ℓ" to its parent. The root detects termination upon receiving "ack" messages from all relevant children.

▶ **Lemma 16.** *Procedure **Downcast**, run by nodes in the tree T, performs downcasting of m distinct messages with termination detection in $O(m + depth(T))$ time and $O(m \cdot depth(T) + size(T))$ messages.*

Finding MOE of a Fragment. Informally, minimum outgoing edge (MOE) out of T is the least weight edge out of T to a node with a different cluster ID (i.e, $\neq C$). Formally, it is a tuple $\langle u, v, C, C' \rangle$ such that edge (u, v) is the MOE from T where $u \in T$ with cluster ID C and $v \notin T$ with cluster ID $C'(\neq C)$. Note that nodes not belonging to T but adjacent to T may have the same cluster ID C as the nodes of T, and as such it is possible for T to not have any MOE. Yet another application of Wave&Echo, taken from the algorithm of [22], results in R being made aware of the MOE of T if such exists. Let us call this module procedure **Find-MOE**.

▶ **Lemma 17.** *Procedure **Find-MOE**, when run by the nodes of a tree T with distinct root R, and cluster ID C, results in R knowing the minimum outgoing edge from T, if one exists, where only edges to nodes with a cluster ID $\neq C$ are considered outgoing edges, in $O(depth(T))$ time and $O(\sum_{u \in T} deg(u))$ messages, where $depth(T)$ is the depth of T and $deg(u)$ is the degree of node u. Furthermore, every node participating in procedure **Find-MOE** can detect termination.*

Size Calculation of a Fragment. We make use of a known tool (essentially a known application of Wave&Echo, see PIF in [54]), to be run by the nodes of the tree and result in R being made aware of how many nodes (including itself) belong to T. Let us call this Procedure **Tree-Count**.

▶ **Observation 18.** *Procedure **Tree-Count**, when run by the nodes of a tree T with distinct root R, results in R knowing the total number of nodes in T in $O(depth(T))$ time and $O(size(T))$ messages, where $depth(T)$ is the depth of T and $size(T)$ is the number of nodes in T. Furthermore, nodes participating in procedure **Tree-Count** can detect termination.*

Diameter Calculation of a Fragment. Another known application of Wave&Echo allows R to calculate the diameter of the tree T, let us call that Procedure **Diam-Calc**.

▶ **Observation 19.** *Procedure **Diam-Calc**, when run by the nodes of a tree T with distinct root R, results in R knowing the diameter of T in $O(depth(T))$ time and $O(size(T))$ messages, where $depth(T)$ is the depth of T and $size(T)$ is the number of nodes in T. Furthermore, nodes participating in procedure **Diam-Calc** can detect termination.*

B Low Diameter Spanning Tree - Relegated Proofs

We first provide definitions and an auxiliary lemma (see Lemma 20) followed by proofs of Lemmas 1 and 2, stated in Section 2.1. After which, we provide the proofs of Lemma 4 and Corollary 5, stated in Section 2.2.

Consider some fixed execution of the algorithm and node $v \in \overline{V}$. Then $D_u = S_u + dist(u,v) - 1 = \delta_{max} - \lfloor \delta_u \rfloor + dist(u,v) - 1$ denotes the *(arrival) round* of u, that is, the first round in which v can receive a message from u's cluster. For every integer $1 \le j \le n$, let z_j be the node with the jth smallest arrival round in the execution. For every integer $1 \le k \le n$, let $S_k = \{z_1, \ldots, z_k\}$. Building upon these definitions, for a node $v \in \overline{V}$, positive integers $1 \le k, r \le n$, let $\mathcal{E}_{v,k,r}$ denote the event that after the execution of the algorithm, $D_{z_{k+1}} - D_{z_1} \le r$.

▶ **Lemma 20.** *For any node $v \in \overline{V}$ and positive integers $1 \le k, r \le n$,*

$$\Pr(\mathcal{E}_{v,k,r}) \le (1 - \exp(-(r+1)\beta))^k$$

Proof. We condition on S_k and $D^* = D_{z_{k+1}}$. The proof is based on first showing the stated upper bound on the probability of $\mathcal{E}_{v,k,r}$ conditioned on S_k and D^*, and then applying the law of total probability to derive the lemma statement. We next describe the first half of the proof in more detail.

For any integer $i \ge 1$, let $c_{z_i} = \delta_{max} + dist(z_i, v) - 1$. We have $\Pr(\mathcal{E}_{v,k,r} \mid S_k, D^*) \le p$ for

$$p = \Pr\left(\bigwedge_{i=1}^{k} [D^* - D_{z_i} \le r]\right) = \Pr\left(\bigwedge_{i=1}^{k} [\delta_{z_i} \le r + 1 + c_{z_i} - D^*]\right) = \prod_{i=1}^{k} \Pr(\delta_{z_i} \le r+1+c_{z_i} - D^*),$$

where the last equality holds since the random variables δ_{z_i} are independent. Next, note that $D^* \ge D_{z_i}$ for any integer $1 \le i \le k$, and thus $\Pr(\lfloor \delta_{z_i} \rfloor \ge c_{z_i} - D^*) = 1$. Hence, $\Pr(\delta_{z_i} \ge c_{z_i} - D^*) = 1$ and

$$p = \prod_{i=1}^{k} \Pr(\delta_{z_i} \le r + 1 + c_{z_i} - D^* \mid \delta_{z_i} \ge c_{z_i} - D^*).$$

Finally,

$$p \le \prod_{i=1}^{k} \Pr(\delta_{z_i} \le r+1) = \prod_{i=1}^{k} (1 - \exp(-(r+1)\beta)) = (1 - \exp(-(r+1)\beta))^k$$

where the inequality holds by the memorylessness of the exponential distribution. ◀

Proof of Lemma 1. We first note for any node $v \in \overline{V}$, $\Pr[\lfloor \delta_v \rfloor > \delta_{max}] = \Pr[\delta_v > \frac{2 \ln n}{\beta}] = \exp(-2 \ln n) = \frac{1}{n^2}$. Hence, by union bound, $\lfloor \delta_v \rfloor \le \delta_{max}$ for every node $v \in \overline{V}$ with high probability. We hereafter exclude this unlikely event and assume $\delta_{max} \ge \max_{v \in \overline{V}} \{\lfloor \delta_v \rfloor\}$. This implies that all nodes belong to a cluster.

Next, note that by the algorithm description, each cluster is spanned by a tree of depth at most $\frac{2 \ln n}{\beta}$. Hence, all clusters have strong diameter at most $\frac{4 \ln n}{\beta}$. Finally, an edge is cut if its two endpoints u and v are in different clusters. This implies that for node v (without loss of generality), the two smallest arrival rounds differ by at most 1, which corresponds to event $\mathcal{E}_{v,1,1}$. By Lemma 20, $\Pr(\mathcal{E}_{v,1,1}) \le (1 - \exp(-2\beta)) \le 2\beta$. The lemma follows. ◀

Proof of Lemma 2. Again, we assume $\delta_{max} \ge \max_{u \in \overline{V}} \{\delta_u\}$, which holds with high probability. For any node $v \in \overline{V}$, let C_v denote the cluster containing v after the execution of the algorithm.

Consider any two nodes $u, v \in \overline{V}$ such that $l = dist_{\overline{G}}(u, v) > 3\beta\overline{D}$. (Note that if $l \leq 3\beta\overline{D}$, then $dist_{\overline{G}^*}(C_u, C_v) \leq 3\beta\overline{D}$.) Let $(w_1, \ldots, w_{l+1})$ be the shortest path between u and v in $\overline{G}$ (where $w_1 = u$ and $w_{l+1} = v$). Moreover, for any integer $i \in [1, l]$, let X_i be the indicator random variable of w_i and w_{i+1} being in the same cluster. Then, the random variable $X = \sum_{i=1}^{l} X_i$ is an upper bound on $dist_{\overline{G}^*}(C_u, C_v)$. By Lemma 1, each edge is an inter-cluster edge with probability at most 2β. Hence, by the linearity of expectation, $E[X] \leq 2\beta l$.

Next, let us provide a concentration bound for X by showing that the random variables X_i are only locally dependent. First, for any two integers $i, j \in [1, l]$ such that $|i - j| > \lfloor 4\frac{\ln n}{\beta} \rfloor$, X_i and X_j are independent (since the same node cannot affect w_i and w_j with our choice of δ_{max}). Then, we can color the random variables $\{X_i\}_{i=1,\ldots,l}$ using $\chi = \lfloor 4\frac{\ln n}{\beta} \rfloor$ – by coloring X_i with $i \mod (\chi + 1)$ – such that variables with the same color are independent. In other words, the random variables X_i are only locally dependent and thus we can apply a specific Chernoff-Hoeffding bound (Theorem 3.2 from [15]): $\Pr(X \geq E[X] + t) \leq \exp(-2t^2/(\chi \cdot l))$. Hence, $\Pr(X \geq 3\beta l) \leq \exp(-2(\beta l)^2/(\chi \cdot l)) \leq \exp(-2\beta^2 l/\chi)$. Since $l > 3\beta\overline{D} > 3k\frac{\ln^2 n}{\beta^3}$, $\Pr(X \geq 3\beta l) \leq \exp(-\frac{3}{2}k \ln n) \leq \frac{1}{n^k}$. By taking a union bound over all n^2 possible pairs of nodes $u, v \in \overline{V}$, the lemma statement follows. ◀

Proof of Lemma 4. By induction on i. The base case, $i = 1$, holds trivially.

Next, consider some $i \geq 1$ for which the inductive hypothesis holds, i.e., $diam(G_{i-1}) = \max\{(3\beta)^{i-1}D, O(\log^{2+4/\varepsilon'} n)\}$ w.h.p. and each cluster node C of the partition $\mathcal{P}_{i-1}$ is spanned (in the original graph G) by a tree $\hat{T}(C)$ with $diam(\hat{T}(C)) = (\frac{5\ln n}{\beta})^{i-1}$. Running Procedure **MPX** on G_{i-1} yields a $(2\beta, \frac{4\ln n}{\beta})$ low-diameter decomposition of G_{i-1}. In fact, each super cluster C' of this decomposition on G_{i-1} is spanned (in the cluster graph G_{i-1}) by a tree $\hat{T}(C')$ of diameter $\frac{4\ln n}{\beta}$. Hence, the "combined" spanning tree computed by Procedure **Transform** for the "analog" C'' of cluster C' on G, which is a cluster of the newly constructed G_i, has diameter $diam(\hat{T}(C'')) = (\frac{4\ln n}{\beta} + 1) \cdot (\frac{5\ln n}{\beta})^{i-1} \leq (\frac{5\ln n}{\beta})^{i}$. Next, the diameter of G_i is the same as that of the cluster graph H induced by partition $\mathcal{P}_i^*$. By Lemma 2, the diameter of H is $\max\{(3\beta)^{i}D, O(\log^{2+4/\varepsilon'} n)\}$ w.h.p., and thus the lemma statement holds. ◀

Proof of Corollary 5. By Lemma 4 (and applying one extra induction step), the diameter of G_{i_m} is $D_f = \max\{(3\beta)^{i_m}D, O(\log^{2+4/\varepsilon'} n)\}$ and each cluster C of the partition $\mathcal{P}_{i_m}$ is spanned in G by a tree $\hat{T}(C)$ of depth $d_f = (\frac{5\ln n}{\beta})^{i_m}$. Since $i_m = \lceil \log_{1/(3\beta)} D \rceil$, we have that $(3\beta)^{i_m} \leq 1/D$, so $D_f = O(\log^{2+4/\varepsilon'} n)$. Moreover, by going through the computations, we get:

$$\begin{aligned} d_f &= \exp(i_m \ln(5\ln^{1+1/\varepsilon'} n)) \;\leq\; (5\ln^{1+1/\varepsilon'} n) \exp\left(\frac{\ln D \ln(5\ln^{1+1/\varepsilon'} n)}{\ln(\frac{1}{3}\ln^{1/\varepsilon'} n)}\right) \\ &= (5\ln^{1+1/\varepsilon'} n) \exp\left(\ln D \cdot \frac{\ln 5 + (1 + 1/\varepsilon')\ln\ln n}{\frac{1}{\varepsilon'}\ln\ln n - \ln 3}\right) \\ &= (5\ln^{1+1/\varepsilon'} n) \exp\left(\ln D \cdot \left(1 + \frac{\ln 5 + \ln 3 + \ln\ln n}{\frac{1}{\varepsilon'}\ln\ln n - \ln 3}\right)\right) \\ &\leq (5\ln^{1+1/\varepsilon'} n) \exp(\ln D \cdot (1 + 2\varepsilon' \ln 15)) \;\leq\; (5\ln^{1+1/\varepsilon'} n)\; D^{1+\varepsilon}\;, \end{aligned}$$

where, in order to make the last inequality hold, Procedure **ST-Cons**(ε) selects $\varepsilon' \leq \varepsilon/(2\ln 15)$. ◀

Locally Restricted Proof Labeling Schemes

Yuval Emek ✉
Technion – Israel Institute of Technology, Haifa, Israel

Yuval Gil ✉
Technion – Israel Institute of Technology, Haifa, Israel

Shay Kutten ✉
Technion – Israel Institute of Technology, Haifa, Israel

Abstract

Introduced by Korman, Kutten, and Peleg (PODC 2005), a *proof labeling scheme (PLS)* is a distributed verification system dedicated to evaluating if a given *configured graph* satisfies a certain property. It involves a centralized *prover*, whose role is to provide proof that a given configured graph is a yes-instance by means of assigning *labels* to the nodes, and a distributed verifier, whose role is to verify the validity of the given proof via local access to the assigned labels. In this paper, we introduce the notion of a *locally restricted* PLS in which the prover's power is restricted to that of a LOCAL algorithm with a polylogarithmic number of rounds. To circumvent inherent impossibilities of PLSs in the locally restricted setting, we turn to models that relax the correctness requirements by allowing the verifier to accept some no-instances as long as they are not "too far" from satisfying the property in question. To this end, we evaluate (1) *distributed graph optimization problems (OptDGPs)* based on the notion of an *approximate proof labeling scheme (APLS)* (analogous to the type of relaxation used in sequential approximation algorithms); and (2) *configured graph families (CGFs)* based on the notion of a *testing proof labeling schemes (TPLS)* (analogous to the type of relaxation used in property testing algorithms). The main contribution of the paper comes in the form of two generic compilers, one for OptDGPs and one for CGFs: given a black-box access to an APLS (resp., PLS) for a large class of OptDGPs (resp., CGFs), the compiler produces a locally restricted APLS (resp., TPLS) for the same problem, while losing at most a $(1 + \epsilon)$ factor in the scheme's relaxation guarantee. An appealing feature of the two compilers is that they only require a logarithmic additive label size overhead.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Approximation algorithms analysis

Keywords and phrases proof labeling schemes, generic compilers, SLOCAL algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.20

Related Version *Full Version*: https://arxiv.org/abs/2208.08718 [8]

Funding This work was supported in part by the Technion Hiroshi Fujiwara Cyber Security Research Center and the Israel National Cyber Directorate. In addition, the work of Shay Kutten was also supported in part by ISF grant 1346/22.

1 Introduction

A *proof system* is a tool designed to verify the correctness of a certain claim. It is composed of two entities: a *prover*, whose role is to provide proof for the claim in question; and a computationally bounded *verifier* that seeks to verify the validity of the given proof. The crux of a proof system is that the proof given by the prover cannot be blindly trusted. That is, for a proof system to be correct, the verifier must be able to distinguish between an honest prover, providing a correct proof, and a malicious prover who tries to convince the verifier of a false claim.

© Yuval Emek, Yuval Gil, and Shay Kutten;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 20; pp. 20:1–20:22
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In the realm of *distributed computing*, the study of proof systems, also known as *distributed proof systems*, has attracted considerable attention. The goal of a distributed proof system is to decide if a given *configured graph* satisfies a certain property. This is typically done by means of a centralized prover, that has a global view of the entire configured graph, and a distributed verifier, that operates at all nodes concurrently and is subject to locality restrictions. Various models for distributed proof systems have been introduced in the literature, including *proof labeling schemes (PLSs)* [19], *locally checkable proofs* [16], *nondeterministic local decisions* [12], and *distributed interactive proofs* [17].

The current paper focuses on the PLS model, introduced by Korman, Kutten, and Peleg [19] (see Sec. 2.1 for the formal definition). In a PLS, the prover generates its proof by means of assigning a *label* to each node. The verification process performed by the verifier at each node v has access to v's label and to the labels of v's neighbors, but it cannot access the labels assigned to nodes outside its local neighborhood. The correctness requirements state that if the given configured graph is a yes-instance, then all nodes must accept; and if the given configured graph is a no-instance, then at least one node must reject. The standard performance measure of a PLS is its *proof size*, defined to be the size of the largest label assigned by the honest prover.

Recently, there is a growing interest from (sequential) computational complexity researchers in *doubly efficient* proof systems [15, 26]. These proof systems are characterized by restricting the (honest) prover to "efficient computations" – i.e., polynomial time algorithms – on top of the restrictions imposed on the computational power of the (still weaker) verifier. For example, Goldwasser et al. [15] consider polynomial time provers vs. logarithmic space verifiers, whereas Reingold et al. [26] consider polynomial time provers vs. linear time and near-linear time verifiers.

Motivated by the success story of doubly efficient proof systems in sequential computational complexity, in this paper, we initiate the study of such proof systems in the distributed computing realm. To do so, we adjust the notion of "efficient computations" from sequential algorithms running in polynomial time to LOCAL algorithms running in a polylogarithmic number of rounds [25]. This introduces a new type of PLSs, called *locally restricted* PLSs, where the label assigned to a node v is computed by the (honest) prover based on the subgraph induced by the nodes within polylogarithmic distance from v, rather than the whole graph (refer to Sec. 2.2 for a formal definition).

Beyond the theoretical interest that lies in this new type of distributed proof systems, we advocate for their investigation also from a more practical point of view: A natural application of PLSs is local checking for self-stabilizing algorithms [2] which involves a detection module and a correction module. In this mechanism, the verifier's role is played by the detection module and the prover's role is played by a dedicated sub-module of the correction module responsible for the label assignment to the nodes [19] (the correction module typically includes another sub-module, responsible for constructing the actual solution, which is abstracted away by the PLS). Since both modules operate as distributed algorithms, any attempt to implement them in practice should take efficiency considerations into account. While classic PLSs consider this efficiency requirement (only) from the verifier's point of view, in locally restricted PLSs, we impose efficiency demands on both the verifier and the prover.

It turns out that locally restricted PLSs are impossible for many interesting properties, regardless of proof size (as shown in the simple observation presented in Appendix B). This leads us to slightly relax the correctness requirements of a PLS so that the verifier may also accept no-instances as long as they are not "too far" from satisfying the property in question. Specifically, we consider locally restricted schemes in the context of two relaxed models called *approximate proof labeling schemes (APLS)* [6, 7] and *testing proof labeling schemes (TPLS)*.

Table 1 Locally restricted APLS (left) and TPLS (right) results, where ℓ stands for the proof size and α stands for the approximation ratio.

OptDGP	graph family	$\boldsymbol{\alpha}$	$\boldsymbol{\ell}$
min. weight vertex cover	any	$2(1+\epsilon)$	$O(\log n)$
min. vertex cover	odd-girth $= \omega(\log n)$	$1+\epsilon$	$O(\log n)$
max. ind. set	any	$\Delta(1+\epsilon)$	$O(\log n)$
	odd-girth $= \omega(\log n)$	$1+\epsilon$	$O(\log n)$
min. weight dom. set	any	$O(\log n)$	$O(\log n)$
any canonical OptDGP	any	$1+\epsilon$	$O(n^2)$

CGF	ℓ
planarity	$O(\log n)$
bounded arboricity	$O(\log n)$
k-colorability	$O(\log n)$
forest	$O(\log n)$
DAG	$O(\log n)$

The APLS model was introduced by Censor-Hillel et al. [6] and studied further by Emek and Gil [7]. For an approximation parameter $\alpha \geq 1$, the goal of an α-APLS for a *distributed graph optimization problem (OptDGP)* is to distinguish between optimal instances and instances that are α-far from being optimal (refer to Sec. 2 for the definitions). Interestingly, for some classic edge-based *covering/packing* OptDGPs (e.g., maximum matching and minimum edge cover), locally restricted APLSs are already established in previous works [16, 6, 7]. In contrast, the existing APLSs for node-based covering/packing OptDGPs require that the prover has a global view of the given configured graph (see, e.g., the APLS for minimum weight vertex cover presented in [7]). In Sec. 4, we develop a generic compiler that gets a (not necessarily locally restricted) α-APLS for an OptDGP Ψ, belonging to a large class of node-based covering/packing OptDGPs, and generates a locally restricted $((1+\epsilon)\alpha)$-APLS for Ψ, where ϵ is a constant performance parameter. The proof size of the locally restricted $((1+\epsilon)\alpha)$-APLS generated by our compiler is $\ell_{\Psi,\alpha} + O(\log n)$, where $\ell_{\Psi,\alpha}$ is the proof size of the α-APLS provided to the compiler. Refer to Section 4.3 for a high-level overview of this construction.

The TPLS model is developed in the current paper based on the notion of *property testing* [14, 1]. For a parameter $\delta > 0$, the goal of a δ-TPLS for a *configured graph family (CGF)* Φ is to distinguish between configured graphs belonging to Φ and configured graphs that are δ-far from belonging to Φ, where the distance here is measured in terms of the graph topology. In Sec. 5, we develop a generic compiler that gets a (not necessarily locally restricted) PLS for a CGF Φ, that is closed under node-induced subgraphs and disjoint union, and generates a locally restricted δ-TPLS for Φ. The proof size of the locally restricted δ-TPLS generated by our compiler is $\ell_\Phi + O(\log n)$, where ℓ_Φ is the proof size of the PLS provided to the compiler. Refer to Section 5.2 for a high-level overview of this construction.

The applicability of our compilers is demonstrated in Appendix A, where we show how the two compilers can be used to obtain APLSs and TPLSs for various well-known OptDGPs and CGFs, respectively; refer to Table 1 for a summary of these results. We conclude with additional related work presented in Appendix C.

1.1 Paper's Organization

In Section 2, we present the model. Preliminaries are presented in Section 3. Following that, in Sections 4 and 5, we present our compiler for OptDGPs and CGFs, respectively. Within these sections, a high-level overview of the compilers appears in Subsections 4.3 and 5.2.

2 Model

We consider *distributed verification systems* in which evaluated instances are called *configured graphs*. A configured graph $G_s = \langle G, s \rangle$ is a pair consisting of an undirected graph $G = (V, E)$ and a *configuration function* $s : V \to \{0,1\}^*$ that assigns a bit string $s(v)$, referred to as v's *local configuration*, to each node $v \in V$. Throughout this paper, we use the notation $n = |V|$ and $m = |E|$.

For a node $v \in V$, we stick to the convention that $N_G(v) = \{u \mid (u,v) \in E\}$ denotes the set of v's *neighbors* in G and that $\deg_G(v) = |N_G(v)|$ denotes v's *degree* in G. When G is clear from the context, we may omit it from our notations and use $N(v)$ and $\deg(v)$ instead of $N_G(v)$ and $\deg_G(v)$, respectively.

We assume that all configured graphs considered in the context of this paper are *identified*, i.e., the configuration function $s : V \to \{0,1\}^*$ assigns a unique id of size $O(\log n)$, denoted by $id(v)$, to each node $v \in V$. Moreover, we assume that the local configuration $s(v)$ distinguishes between node v's incident edges by means of a set $\mathcal{A}(v)$ of abstract *port names*, and a bijection $\rho_v^s : N(v) \to \mathcal{A}(v)$, referred to as the *internal port name assignment* of v, that assigns a (locally unique) port name $\rho_v^s(u)$ to each node $u \in N(v)$. More concretely, assume that the local configuration $s(v)$ includes a designated field for each neighbor $u \in N(v)$ and that this field is indexed by $\rho_v^s(u)$. Unless stated otherwise, when we refer to an ordered list $u_1, \dots, u_{\deg(v)}$ of v's neighbors, it is assumed that the list is ordered by v's internal port name assignment.

Given a configured graph G_s consisting of graph $G = (V, E)$ and configuration function s, we say that a configured graph $G'_{s'}$ consisting of graph $G' = (V', E')$ and configuration function s', is a *configured subgraph* of G_s if (1) G' is a subgraph of G, i.e., $V' \subseteq V$ and $E' \subseteq E$; and (2) the configuration function s' is the projection of s on G', where for each node $v \in V'$, the fields corresponding to nodes $u \in N_G(v) \setminus N_{G'}(v)$ are omitted from the local configuration $s'(v)$ and the internal port name assignment $\rho_v^{s'}$ associated with s' is defined so that $\rho_v^{s'}(u) = \rho_v^s(u)$ for each $u \in N_{G'}(v)$. For a node subset $U \subseteq V$, let $G(U)$ denote the subgraph induced on G by U and let $G_s(U)$ be the configured subgraph of G_s defined over the subgraph $G(U)$.

We define a *configured graph family (CGF)* as a collection of configured graphs. A CGF type that plays a central role in this paper is that of a *distributed graph problem (DGP)* Π, where for each configured graph $G_s \in \Pi$, the configuration function s is composed of an *input assignment* $\mathsf{i} : V \to \{0,1\}^*$ and an *output assignment* $\mathsf{o} : V \to \{0,1\}^*$. We refer to such a configured graph as an *input-output (IO) graph* and often denote it by $G_{\mathsf{i},\mathsf{o}}$. The input assignment i assigns to each node $v \in V$, a bit string $\mathsf{i}(v)$, referred to as v's *local input*, that encodes attributes associated with v and its incident edges (e.g., node ids, edge orientations, edge weights, and node weights); whereas the output assignment o assigns a *local output* $\mathsf{o}(v)$ to each node $v \in V$. For an input assignment i, we refer to the configured graph $G_{\mathsf{i}} = \langle G, \mathsf{i} \rangle$ as an *input graph.*

Consider a DGP Π. An input graph G_{i} is said to be *legal* (and the graph G and input assignment i are said to be *co-legal*) if there exists an output assignment o such that $G_{\mathsf{i},\mathsf{o}} \in \Pi$, in which case we say that o is a *feasible solution* for G_{i} (or simply for G and i). For a DGP Π, we denote the set of legal input graphs by $\mathcal{LEG}(\Pi) = \{G_{\mathsf{i}} \mid \exists \mathsf{o} : G_{\mathsf{i},\mathsf{o}} \in \Pi\}$.

A *distributed graph minimization problem (MinDGP)* (resp., *distributed graph maximization problem (MaxDGP)*) Ψ is a pair $\langle \Pi, f \rangle$, where Π is a DGP and $f : \Pi \to \mathbb{Z}$ is a function, referred to as the *objective function* of Ψ, that maps each IO graph $G_{\mathsf{i},\mathsf{o}} \in \Pi$ to an integer value $f(G_{\mathsf{i},\mathsf{o}})$.[1] Given a co-legal graph G and input assignment i, define

[1] We assume for simplicity that the images of the objective functions used in the context of this paper are

$$OPT_\Psi(G, \mathsf{i}) = \inf_{\mathsf{o}:G_{\mathsf{i},\mathsf{o}} \in \Pi} \{f(G_{\mathsf{i},\mathsf{o}})\}$$

if Ψ is a MinDGP; and

$$OPT_\Psi(G, \mathsf{i}) = \sup_{\mathsf{o}:G_{\mathsf{i},\mathsf{o}} \in \Pi} \{f(G_{\mathsf{i},\mathsf{o}}\}$$

if Ψ is a MaxDGP. We often use the general term *distributed graph optimization problem (OptDGP)* to refer to MinDGPs as well as MaxDGPs. Given an OptDGP $\Psi = \langle \Pi, f \rangle$ and co-legal graph G and input assignment i, the output assignment o is said to be an *optimal solution* for G_i (or simply for G and i) if o is a feasible solution for G_i and $f(G_{\mathsf{i},\mathsf{o}}) = OPT_\Psi(G, \mathsf{i})$.

2.1 Proof Labeling Schemes

In this section we present the notion of proof labeling schemes as well as its approximation variants. To that end, we first present the notion of gap proof labeling schemes, as defined in [7].

Fix some universe $\mathcal{U}$ of configured graphs. A *gap proof labeling scheme (GPLS)* is a mechanism designed to distinguish the configured graphs in a *yes-family* $\mathcal{F}_Y \subset \mathcal{U}$ from the configured graphs in a *no-family* $\mathcal{F}_N \subset \mathcal{U}$, where $\mathcal{F}_Y \cap \mathcal{F}_N = \emptyset$. This is done by means of a (centralized) *prover* and a (distributed) *verifier* that play the following roles: Given a configured graph $G_s \in \mathcal{U}$, if $G_s \in \mathcal{F}_Y$, then the prover assigns a bit string $L(v)$, called the *label* of v, to each node $v \in V$. Let $L^N(v) = \langle L(u_1), \ldots, L(u_{\deg(v)}) \rangle$ be the vector of labels assigned to v's neighbors. The verifier at node $v \in V$ is provided with the 3-tuple $\langle s(v), L(v), L^N(v) \rangle$ and returns a Boolean value $\varphi(v)$.

We say that the verifier *accepts* G_s if $\varphi(v) =$ `True` for all nodes $v \in V$; and that the verifier *rejects* G_s if $\varphi(v) =$ `False` for at least one node $v \in V$. The GPLS is said to be *correct* if the following requirements hold for every configured graph $G_s \in \mathcal{U}$:

▶ **R1.** *If $G_s \in \mathcal{F}_Y$, then the prover produces a label assignment $L : V \to \{0,1\}^*$ such that the verifier accepts G_s.*

▶ **R2.** *If $G_s \in \mathcal{F}_N$, then for any label assignment $L : V \to \{0,1\}^*$, the verifier rejects G_s.*

We emphasize that no requirements are made for configured graphs $G_s \in \mathcal{U} \setminus (\mathcal{F}_Y \cup \mathcal{F}_N)$; in particular, the verifier may either accept or reject these configured graphs (the same holds for configured graphs that do not belong to the universe $\mathcal{U}$). The performance of a GPLS is measured by means of its *proof size* defined to be the maximum length of a label $L(v)$ assigned by the prover to the nodes $v \in V$ assuming that $G_s \in \mathcal{F}_Y$.

Proof Labeling Schemes for CGFs. Consider some CGF Φ and let $\mathcal{U}$ be the universe of all configured graphs. A *proof labeling scheme (PLS)* for Φ is the GPLS over $\mathcal{U}$ defined by setting the yes-family to be $\mathcal{F}_Y = \Phi$; and the no-family to be $\mathcal{F}_N = \mathcal{U} \setminus \mathcal{F}_Y$. In other words, a PLS for Φ determines whether a given configured graph G_s belongs to Φ.

integral. Lifting this assumption and allowing for real numerical values would complicate some of the arguments, but it does not affect the validity of our results.

In this paper, we also define a relaxed model of PLSs for a CGF Φ in which we allow the verifier to accept configured graphs that are not "too far" from belonging to Φ. To that end, we use the following distance measure which is widely used in the realm of property testing (see e.g., [1]).

let G_s and $G'_{s'}$ be two configured graphs. Given a parameter $\delta > 0$, we say that G_s and $G'_{s'}$ are *δ-close* if $G'_{s'}$ is a configured subgraph of G_s and G' can be obtained from G by removing at most δm edges (or vice versa).

Consider a CGF Φ. We say that a configured graph G_s is *δ-far* from belonging to Φ if $G'_{s'} \notin \Phi$ for any configured graph $G'_{s'}$ which is δ-close to G_s. We define a *δ-testing proof labeling scheme (δ-TPLS)* in the same way as a PLS for Φ with the sole difference that the no-family is defined by setting $\mathcal{F}_N = \{G_s \mid G_s \text{ is } \delta\text{-far from belonging to } \Phi\}$.

Proof Labeling Schemes for OptDGPs. Consider some OptDGP $\Psi = \langle \Pi, f \rangle$ and let $\mathcal{U} = \{G_{\mathsf{i},\mathsf{o}} \mid G_{\mathsf{i}} \in \mathcal{LEG}(\Pi)\}$. A *proof labeling scheme (PLS)* for Ψ is defined as a GPLS over $\mathcal{U}$ by setting the yes-family to be

$$\mathcal{F}_Y = \{G_{\mathsf{i},\mathsf{o}} \in \Pi \mid f(G_{\mathsf{i},\mathsf{o}}) = OPT_\Psi(G, \mathsf{i})\}$$

and the no-family to be $\mathcal{F}_N = \mathcal{U} \setminus \mathcal{F}_Y$. In other words, a PLS for Ψ determines for a given IO graph $G_{\mathsf{i},\mathsf{o}} \in \mathcal{U}$ whether the output assignment $\mathsf{o} : V \to \{0,1\}^*$ is an optimal solution (which means in particular that it is a feasible solution) for the co-legal graph $G = (V, E)$ and input assignment $\mathsf{i} : V \to \{0,1\}^*$.

In the realm of OptDGPs, a relaxed model called approximate proof labeling scheme has been considered in [6, 7]. In this model, the correctness requirement of a PLS are relaxed so that it may also accept feasible solutions that only approximate the optimal ones. Specifically, given an approximation parameter $\alpha \geq 1$, an *α-approximate proof labeling scheme (α-APLS)* for an OptDGP $\Psi = \langle \Pi, f \rangle$ is defined in the same way as a PLS for Ψ with the sole difference that the no-family is defined by setting

$$\mathcal{F}_N = \begin{cases} \mathcal{U} \setminus \{G_{\mathsf{i},\mathsf{o}} \in \Pi \mid f(G_{\mathsf{i},\mathsf{o}}) \leq \alpha \cdot OPT_\Psi(G, \mathsf{i})\}\ , & \text{if } \Psi \text{ is a MinDGP} \\ \mathcal{U} \setminus \{G_{\mathsf{i},\mathsf{o}} \in \Pi \mid f(G_{\mathsf{i},\mathsf{o}}) \geq OPT_\Psi(G, \mathsf{i})/\alpha\}\ , & \text{if } \Psi \text{ is a MaxDGP} \end{cases} .$$

2.2 Locally Restricted Proof Labeling Schemes

In this paper, we focus on provers whose power is limited as follows. We say that a GPLS is *locally restricted* if there exists a constant c such that for every configured graph $G_s = \langle G = (V, E), s \rangle \in \mathcal{F}_Y$ and for every node $v \in V$, the label $L(v)$ is computed by the prover as a function of $G_s(B^r(v))$, where $r = \log^c n$ and $B^r(v)$ denotes the set of nodes at (hop) distance at most r from v in G. Equivalently, the prover is restricted to a distributed algorithm operating under the LOCAL model [22, 25] with polylogarithmic rounds. We emphasize that if $G_s \in \mathcal{F}_N$, then the verifier is required to reject G_s for any label assignment, including label assignments that were not produced in a locally restricted fashion.

3 Preliminaries

Sequentially Local Algorithms. In the *sequentially local (SLOCAL)* model, introduced in [13], each node $v \in V$ maintains two (initially empty) bit strings denoted by $\text{info}(v)$ and $\text{decision}(v)$. Nodes are processed sequentially in an arbitrary order $p = v_1, \ldots, v_n$ (i.e., irrespective of node ids). We refer to the time that node v_i is processed as the i-th iteration

of the algorithm. In the i-th iteration, v_i has a read/write access to info(u) for all nodes $u \in B^r(v_i)$, where $r \in \mathbb{Z}_{\geq 0}$ is a parameter referred to as the *locality* of the algorithm. Following that, v_i writes an irrevocable value into decision(v_i) based strictly on $G_s(B^r(v_i))$ and the bit strings info(u) of all $u \in B^r(v_i)$.

A consequence of the seminal work of Ghafari et al. [13, 27] is that any SLOCAL algorithm with $\log^{O(1)} n$ locality can be simulated by a LOCAL algorithm with $\log^{O(1)} n$ rounds. Therefore, in the context of a locally restricted GPLS, by allowing the prover to compute the label $L(v)$ of each node $v \in V$ using an SLOCAL algorithm with locality $r = \log^{O(1)} n$ (rather than a LOCAL algorithm with $\log^{O(1)} n$ rounds), we do not increase the scheme's power.

Comparison Schemes. Let $\mathcal{U}$ be a universe of configured graphs $G_{s_{a,b}}$, such that $G = (V, E)$ is a connected undirected graph and the configuration function $s_{a,b} : V \to \{0,1\}^*$ assigns two values $a(v), b(v) \in \mathbb{R}$ to each node $v \in V$. A *comparison scheme* is a mechanism whose goal is to decide if $\sum_{v \in V} a(v) \geq \sum_{v \in V} b(v)$ for a given configured graph $G_{s_{a,b}} \in \mathcal{U}$. Formally, a comparison scheme is defined as a GPLS over $\mathcal{U}$ by setting the yes-family to be $\mathcal{F}_Y = \{G_{s_{a,b}} \in \mathcal{U} \mid \sum_{v \in V} a(v) \geq \sum_{v \in V} b(v)\}$; and the no family to be $\mathcal{F}_N = \mathcal{U} \setminus \mathcal{F}_Y$.

In [19, Lemma 4.4], Korman et al. present a generic design for comparison schemes as follows. Consider a configured graph $G_{s_{a,b}} \in \mathcal{U}$. The label assignment $L : V \to \{0,1\}^*$ constructed by the prover encodes a spanning tree of G rooted at some (arbitrary) node $r \in V$ (see [19, Lemma 2.2] for details on spanning tree construction). In addition, the prover encodes the sum of $a(\cdot)$ and $b(\cdot)$ values in the sub-tree rooted at node v for each $v \in V$. This allows the verifier to check that the sums assigned at each node $v \in V$ are correct (using the sums assigned to v's children); and the verifier at the root r can evaluate if $G_{s_{a,b}} \in \mathcal{F}_Y$. The proof size of this scheme is $O(\log n + M_{a,b})$, where $M_{a,b}$ is the maximum length (in bits) of values $\sum_{v \in U} a(v)$ and $\sum_{v \in U} b(v)$ over all node-subsets $U \subseteq V$. This comparison scheme construction is used as an auxiliary tool in the compilers presented in Sec. 4 and 5.

4 Compiler for OptDGPs

In this section, we present our generic compiler for OptDGPs. It is divided into five subsections as follows. First, in Sec. 4.1 we characterize the OptDGPs that are suited for our compiler, referred to as canonical OptDGPs, based on the notions of locally checkable labelings and covering/packing OptDGPs (these terms are formally defined in Sec. 4.1). In Sec. 4.2, we establish an important property of optimal solutions for covering/packing OptDGPs that serve the compiler construction. Sec. 4.4 and 4.5 are dedicated to the compiler construction. More formally, these sections constructively prove the following theorem.

▶ **Theorem 4.1.** *Let Ψ be a canonical OptDGP that admits an α-APLS with a proof size of $\ell_{\Psi,\alpha}$. For any constant $\epsilon > 0$, there exists a locally restricted $(\alpha(1+\epsilon))$-APLS for Ψ with a proof size of $\ell_{\Psi,\alpha} + O(\log n)$.*

For convenience, the compiler construction is divided between Sec. 4.4 and 4.5 as follows. In Sec. 4.4, we present an SLOCAL algorithm with logarithmic locality that partitions the nodes into disjoint clusters, such that the subgraph induced by each cluster is of logarithmic diameter. The goal of this partition is to enable the prover to construct the label of a node as a function of the subgraph induced by its cluster (and possibly some nodes adjacent to its cluster) without information on nodes that are farther away. This partition facilitates the label assignment and verification process described in Sec. 4.5. In Sec. 4.3, we provide a high-level overview of the SLOCAL algorithm and how it is used in the label assignment and verification process.

4.1 Canonical OptDGPs

Locally Checkable Labelings. A DGP Π is said to be a *locally checkable labeling (LCL)* (cf. [23]) if there exists a Boolean predicate family $\mathcal{L}^{\Pi} = \{p^{\Pi}_{d,\ell} : (\{0,1\}^*)^{d+1} \rightarrow \{\texttt{True}, \texttt{False}\}\}_{d \in \mathbb{Z}_{\geq 0}, \ell \in \{0,1\}^*}$ such that for every legal input graph $G_{\mathsf{i}} \in \mathcal{LEG}(\Pi)$, an output assignment $\mathsf{o} : V \rightarrow \{0,1\}^*$ is a feasible solution for G and i if and only if $p^{\Pi}_{\deg(v),\mathsf{i}(v)}(\mathsf{o}(v), \mathsf{o}(u_1), \ldots, \mathsf{o}(u_{\deg(v)})) = \texttt{True}$ for every node $v \in V$ with neighbors $u_1, \ldots, u_{\deg(v)}$.

For convenience, we assume that the local input $\mathsf{i}(v)$ of a node $v \in V$ is partitioned into two fields, denoted by $\texttt{prd}(\mathsf{i}(v))$ and $\texttt{data}(\mathsf{i}(v))$, where the former (fully) determines the predicate $p^{\Pi}_{\deg(v),i(v)}$ associated with $\deg(v)$ and $\mathsf{i}(v)$ and the latter encodes all other pieces of information included in $\mathsf{i}(v)$. This allows us to slightly abuse the notation and write $p^{\Pi}_{\texttt{prd}(\mathsf{i}(v))}$ instead of $p^{\Pi}_{\deg(v),\mathsf{i}(v)}$. We further assume that the Boolean predicate family $\mathcal{L}^{\Pi}$ includes the trivial tautology predicate $taut_d : (\{0,1\}^*)^{d+1} \rightarrow \{\texttt{True}, \texttt{False}\}$, $d \in \mathbb{Z}_{\geq 0}$, that satisfies $taut_d(x) = \texttt{True}$ for every $x \in (\{0,1\}^*)^{d+1}$ and that this predicate is encoded by writing the designated bit string $taut_d$ in the $\texttt{prd}(\cdot)$ field of the local input.

We say that an LCL Π is *self-induced* if the following two conditions are satisfied for every legal input graph $G_{\mathsf{i}} \in \mathcal{LEG}(\Pi)$ and node subset $U \subseteq V$: (1) $G_{\mathsf{i}}(U) \in \mathcal{LEG}(\Pi)$; and (2) if $\mathsf{i}' : V \rightarrow \{0,1\}^*$ is the input assignment derived from i by setting $\texttt{data}(\mathsf{i}'(v)) = \texttt{data}(\mathsf{i}(v))$ and $\texttt{prd}(\mathsf{i}'(v)) = taut_{\deg(v)}$ for every $v \in U$, then $G_{\mathsf{i}'} \in \mathcal{LEG}(\Pi)$.

Let Π be an LCL and let $G_{\mathsf{i}} \in \mathcal{LEG}(\Pi)$. For a subset $U \subseteq V$ of nodes, we denote by $\texttt{inner}(U) = \{u \in U \mid N_G(u) \subseteq U\}$ the set of nodes in U for which every neighbor is in U and define $\texttt{inner}^2(U) = \texttt{inner}(\texttt{inner}(U))$ and $\texttt{rim}(U) = U \setminus \texttt{inner}^2(U)$. We say that a function $g : U \rightarrow \{0,1\}^*$ *respects* Π if $p^{\Pi}_{\texttt{prd}(\mathsf{i}(v))}(g(v), g(u_1), \ldots, g(u_{\deg(v)})) = \texttt{True}$ for each $v \in \texttt{inner}(U)$ with neighbors $u_1, \ldots, u_{\deg(v)}$.

Canonical OptDGPs. A MinDGP (resp., MaxDGP) $\Psi = \langle \Pi, f \rangle$ is said to be a *covering* (resp., *packing*) OptDGP if the following conditions hold: (1) Π is an LCL; (2) for each n-node IO graph $G_{\mathsf{i},\mathsf{o}} \in \Pi$, there exists a positive integer $k = k(\Pi, n) = n^{O(1)}$ such that the output assignment o assigns a nonnegative integer $\mathsf{o}(v) \in \{0, \ldots, k\}$, referred to as v's *multiplicity*, to each node $v \in V$; (3) for each predicate $p^{\Pi}_{d,\ell} \in \mathcal{L}^{\Pi}$, if $p^{\Pi}_{d,\ell}(x_0, x_1, \ldots, x_d) = \texttt{True}$ for nonnegative integers $x_0, x_1, \ldots, x_d \in \{0, \ldots, k\}$, then $p^{\Pi}_{d,\ell}(x'_0, x'_1, \ldots, x'_d) = \texttt{True}$ for any nonnegative integers $x'_0, x'_1, \ldots, x'_d \in \{0, \ldots, k\}$ that satisfy $x'_j \geq x_j$ (resp., $x'_j \leq x_j$) for all $0 \leq j \leq d$; (4) for every legal input graph $G_{\mathsf{i}} \in \mathcal{LEG}(\Pi)$, there exists a node-weight function $w : V \rightarrow \{1, \ldots n^{O(1)}\}$ such that the weight $w(v)$ of node v is encoded in v's local input field $\texttt{data}(\mathsf{i}(v))$; and (5) $f(G_{\mathsf{i},\mathsf{o}}) = \sum_{v \in V} w(v) \cdot \mathsf{o}(v)$ for every $G_{\mathsf{i},\mathsf{o}} \in \Pi$. The OptDGP $\Psi = \langle \Pi, f \rangle$ is said to be *canonical* if it is covering/packing and Π is self-induced.

Consider a covering/packing OptDGP $\Psi = \langle \Pi, f \rangle$. Let $G_{\mathsf{i}} \in \mathcal{LEG}(\Pi)$ be a legal input graph with the underlying node-weight function $w : V \rightarrow \{1, \ldots, n^{O(1)}\}$ and let $U \subseteq V$. Given a function $g : U \rightarrow \{0, \ldots, k(\Pi, n)\}$ that assigns a multiplicity value $g(u)$ to each node $u \in U$, we define $w(U, g) = \sum_{u \in U} w(u) \cdot g(u)$.

For a covering MinDGP Ψ, let $w_{\min}(U)$ denote the minimum possible value of $w(U, g)$ obtained by a function $g : U \rightarrow \{0, \ldots, k(\Pi, n)\}$ that respects Π. Let $N^2(U)$ be the set of nodes in $V \setminus U$ at distance at most 2 from a node in U. For a packing MaxDGP, let $w_{\max}(U)$ denote the maximum possible value of $w(U, g)$ obtained by a function $g : U \cup N^2(U) \rightarrow \{0, \ldots, k(\Pi, n)\}$ that satisfies (1) $g(v) = 0$ for each node $v \in N^2(U)$; and (2) g respects Π.

4.2 Properties of Optimal Solutions for Covering/Packing OptDGPs

In the following lemmas, we establish important properties regarding optimal solutions of covering and packing OptDGPs. Consider a covering (resp., packing) OptDGP $\Psi = \langle \Pi, f \rangle$. Let $G_{\mathsf{i},\mathsf{o}} \in \Pi$ be an IO graph such that $\mathsf{o} : V \to \{0, \ldots, k(\Pi, n)\}$ is an optimal solution for G and i.

▶ **Lemma 4.2.** *If Ψ is a covering MinDGP, then $w(\mathtt{inner}^2(U), \mathsf{o}) \leq w_{\min}(U)$ for any $U \subseteq V$.*

Proof. Assume by contradiction that $w(\mathtt{inner}^2(U), \mathsf{o}) > w_{\min}(U)$ for some $U \subseteq V$. This means that there exists an assignment $\mathsf{o}' : U \to \{0, \ldots, k(\Pi, n)\}$ that respects Π, such that $w(\mathtt{inner}^2(U), \mathsf{o}) > w(U, \mathsf{o}')$. Let $\tilde{\mathsf{o}}$ be the output assignment defined as follows: $\tilde{\mathsf{o}}(v) = \mathsf{o}(v)$ for all $v \in V \setminus U$; $\tilde{\mathsf{o}}(v) = \mathsf{o}'(v)$ for all $v \in \mathtt{inner}^2(U)$; and $\tilde{\mathsf{o}}(v) = \max\{\mathsf{o}(v), \mathsf{o}'(v)\}$ for all $v \in \mathtt{rim}(U)$. Recall that o is a feasible solution for G and i and that o' respects Π. Since Ψ is a covering OptDGP, we get that $p^{\Pi}_{\mathtt{prd}(\mathsf{i}(v))}(\tilde{\mathsf{o}}(v), \tilde{\mathsf{o}}(u_1), \ldots, \tilde{\mathsf{o}}(u_{\deg_G(v)})) = \mathtt{True}$ for each node $v \in V$ (where $u_1, \ldots, u_{\deg_G(v)}$ denote v's neighbors in G). It follows that $\tilde{\mathsf{o}}$ is a feasible solution with objective value $f(G_{\mathsf{i},\tilde{\mathsf{o}}}) = w(V, \tilde{\mathsf{o}}) \leq w(V \setminus U, \mathsf{o}) + w(\mathtt{inner}^2(U), \mathsf{o}') + w(\mathtt{rim}(U), \mathsf{o}) + w(\mathtt{rim}(U), \mathsf{o}') = w(V, \mathsf{o}) - w(\mathtt{inner}^2(U), \mathsf{o}) + w(U, \mathsf{o}') < w(V, \mathsf{o}) = f(G_{\mathsf{i},\mathsf{o}})$ which contradicts the optimality of o. ◀

▶ **Lemma 4.3.** *If Ψ is a packing MaxDGP, then $w(U, \mathsf{o}) \geq w_{\max}(\mathtt{inner}^2(U))$ for any $U \subseteq V$.*

The proof of Lemma 4.3 is similar to the proof for Lemma 4.2. We defer it to the full version of this paper [8].

4.3 Overview

In Sec. 4.4, we present an SLOCAL algorithm called Part_OPT that partitions the nodes of a given IO graph $G_{\mathsf{i},\mathsf{o}}$ into clusters. Before formally describing the algorithm in Sec. 4.4, let us provide some intuition by presenting the high-level idea of the partition and how it is used in the label and verification process (as described in Sec. 4.5) for the case of a canonical MinDGP Ψ (the high-level idea for MaxDGPs is similar and the differences are mostly technical).

Given an IO graph $G_{\mathsf{i},\mathsf{o}}$, where o is an optimal solution, we use a ball growing argument to obtain a partition of the nodes into clusters such that: (1) the subgraph induced by each cluster is of logarithmic diameter; and (2) the total weight of nodes in the rim of clusters (i.e., nodes with distance at most 2 from a different cluster) is an ϵ-fraction of the total weight of inner nodes of clusters (i.e., nodes with distance at least 3 from a different cluster).

The goal of the partition obtained by Part_OPT is to allow the prover to compute the label assigned to each node based on its cluster. Essentially, the prover seeks to provide the verifier with a proof that the partition satisfies 2 main properties: (1) for each cluster V_j, the weight of the given solution induced on the inner nodes is at most the weight of an approximately optimal (global) solution induced on the cluster; and (2) the total weight of nodes in the rim of clusters is at most an ϵ-fraction of the total weight of inner nodes.

While providing proof for the first property is rather straightforward using the labels of an α-APLS for Ψ in a black box manner, providing proof for the second property is somewhat more challenging. The reason is that the property as presented above is rather global – not every cluster is guaranteed to have at most an ϵ-fraction of its weight assigned to the rim nodes. Constructing a label that sums the total weights of rim and inner nodes of all clusters is a global task and can not be accomplished in a locally restricted fashion. To that end, during Part_OPT, we may assign some nodes with a *secondary affiliation* to an

adjacent cluster. The idea is that for each cluster V_j, the sum between weights of nodes with secondary affiliation to V_j and nodes in $\texttt{rim}(V_j)$ that do not have a secondary affiliation to any cluster is bounded by an ϵ-fraction of V_j's inner nodes weight.

Throughout `Part_OPT`, each node v maintains a *color* whose role is to keep track of the changeability status of v's secondary affiliation. The color white indicates that the secondary affiliation may still change; whereas black indicates that the secondary affiliation is final.

4.4 Partition Algorithm

Algorithm's Description. We now provide a formal description of the `Part_OPT` algorithm. Consider a canonical OptDGP $\Psi = \langle \Pi, f \rangle$. Let $G_{\mathsf{i},\mathsf{o}} \in \Pi$ be an IO graph such that o is an optimal solution for G and i. The algorithm partitions the nodes of G into (possibly empty) clusters $V = V_1 \dot{\cup} \ldots \dot{\cup} V_n$. As usual in the SLOCAL model, the nodes are processed sequentially in n iterations based on an arbitrary order $v_1, \ldots, v_n$ on the nodes, where node v_j is processed in the j-th iteration.

Throughout the execution of `Part_OPT`, each node $v \in V$ maintains three fields referred to as cluster(v), sec(v), and color(v). The field cluster(v) is initially empty and its role is to identify v's cluster, where each cluster V_j is identified by the id of node v_j which is processed in the j-th iteration, i.e., $V_j = \{v \mid \text{cluster}(v) = id(v_j)\}$. The field sec($v$) is initially empty and its role is to identify v's *secondary affiliation* to a cluster if such affiliation exists (otherwise it remains empty throughout the algorithm). The field $\text{color}(v) \in \{\text{black}, \text{white}\}$, initially set to white, maintains v's *color*.

We describe the j-th iteration of `Part_OPT` as follows. Let G_j be the subgraph induced on G by $V \setminus (V_1 \cup \cdots \cup V_{j-1})$. If $v_j \in V_1 \cup \cdots \cup V_{j-1}$, then we define $V_j = \emptyset$ and finish the iteration; so, assume that v_j is a node in G_j. For an integer $r \in \mathbb{Z}_{\geq 0}$, let D_j^r be the set of nodes at distance exactly r from v_j in G_j and let $B_j^r = \bigcup_{r'=0}^{r} D_j^{r'}$. Let $white_j$ be the set of nodes in G_j that are colored white in the beginning of the j-th iteration.

Suppose that Ψ is a MinDGP. We define $r(j)$ to be the smallest integer that satisfies $w(\texttt{inner}^2(B_j^{r(j)+6}), \mathsf{o}) \leq (1+\epsilon) \cdot w(\texttt{inner}^2(B_j^{r(j)+2}), \mathsf{o})$. Notice that $\texttt{inner}^2(\cdot)$ is taken with respect to nodes in G (and not G_j), i.e., $\texttt{inner}^2(B_j^{r(j)+6})$ (resp., $\texttt{inner}^2(B_j^{r(j)+2})$) is the set of nodes in $B_j^{r(j)+6}$ (resp., $B_j^{r(j)+2}$) for which every node within distance 2 in G is in $B_j^{r(j)+6}$ (resp., $B_j^{r(j)+2}$). In the case that Ψ is a MaxDGP, define $r(j)$ to be the smallest integer that satisfies $w(B_j^{r(j)+6}, \mathsf{o}) \leq (1+\epsilon) \cdot w(B_j^{r(j)+2}, \mathsf{o})$.

Following the computation of $r(j)$, we define the cluster V_j and modify the color and secondary affiliation of some nodes as follows (this process is the same for MinDGPs and MaxDGPs). Let X_j be the set of white nodes in $\texttt{inner}^2(B_j^{r(j)+6})$ at distance exactly $r(j)+3$ from v_j, and let Y_j be the set of white nodes in $\texttt{inner}^2(B_j^{r(j)+6})$ at distance exactly $r(j)+4$ from v_j that have a neighbor in X_j. We complete the j-th iteration by setting $\text{cluster}(v) = id(v_j)$ for each node $v \in B_j^{r(j)+2}$ (i.e., setting $V_j = B_j^{r(j)+2}$); $\text{sec}(v) = id(v_j)$ for each node $v \in X_j \cup Y_j$; and $\text{color}(v) = \text{black}$ for each node $v \in X_j$.

Algorithm's Properties. We go on to analyze some properties of `Part_OPT`. Consider a cluster V_j. Let $\texttt{sec}(V_j) = \{v \mid \text{sec}(v) = id(v_j)\}$ be the set of nodes whose secondary affiliation is to V_j by the end of the algorithm and let $\texttt{ext}(V_j) = V_j \cup \texttt{sec}(V_j)$.

▶ **Lemma 4.4.** *The subgraphs $G(V_j)$ and $G(\texttt{ext}(V_j))$ induced on G by V_j and $\texttt{ext}(V_j)$, respectively, are connected and have diameter $O(\log n)$ for each $j \in [n]$.*

Proof. Suppose that $V_j \neq \emptyset$ (as the lemma is trivial otherwise). First, observe that by definition, all nodes $v \in V_j$ are reachable from v_j in $G(V_j)$, thus $G(V_j)$ is connected.

To see that $G(\mathtt{ext}(V_j))$ is connected, we first observe that the subgraph $G(V_j \cup X_j \cup Y_j)$ is connected. By the time cluster V_j is determined, we color the nodes of X_j black. Thus, their secondary affiliation remains to V_j throughout the algorithm. At termination, it follows that $\mathtt{ext}(V_j) = V_j \cup X_j \cup Y$ for some $Y \subseteq Y_j$. Since the nodes of Y all have a neighbor in X_j, we get that $G(\mathtt{ext}(V_j)) = G(V_j \cup X_j \cup Y)$ is connected.

To show that the diameters of $G(V_j)$ and $G(\mathtt{ext}(V_j))$ are $O(\log n)$ it is sufficient to show that $r(j) = O(\log n)$. We use a ball growing argument. By definition, for every $r' < r(j) + 6$, it holds that

$$w(\mathtt{inner}^2(B_j^{r(j)+6}), \mathtt{o}) \geq w(\mathtt{inner}^2(B_j^{r'}), \mathtt{o}) > (1+\epsilon) \cdot w(\mathtt{inner}^2(B_j^{r'-4}), \mathtt{o}) > (1+\epsilon)^2 \cdot w(\mathtt{inner}^2(B_j^{r'-8}), \mathtt{o}) > \dots$$

if Ψ is a MinDGP; and

$$w(B_j^{r(j)+6}, \mathtt{o}) \geq w(B_j^{r'}, \mathtt{o}) > (1+\epsilon) \cdot w(B_j^{r'-4}, \mathtt{o}) > (1+\epsilon)^2 \cdot w(B_j^{r'-8}, \mathtt{o}) > \dots$$

if Ψ is a MaxDGP. Since the terms $w(\mathtt{inner}^2(B_j^{r(j)+6}), \mathtt{o})$ and $w(B_j^{r(j)+6}, \mathtt{o})$ are both bounded by a polynomial of n, it follows that $r(j) = O((1/\epsilon) \log n) = O(\log n)$ in both cases. ◀

A simple observation derived from Lemma 4.4 is that Part_OPT has locality $O(\log n)$. This observation combined with the results of [13, 27] lead to the following corollary.

▶ **Corollary 4.5.** *The algorithm* Part_OPT *can be simulated by a LOCAL algorithm with polylogarithmic round-complexity.*

For each $j \in [n]$, define $S_j = \mathtt{sec}(V_j) \cup \{v \in \mathtt{rim}(V_j) \mid \mathrm{sec}(v) \text{ is empty}\}$ as the set of nodes composed of nodes outside of V_j whose secondary affiliation is to V_j and nodes in $\mathtt{rim}(V_j)$ that do not have a secondary affiliation. The following observation establishes an important property regarding the sets $\mathtt{rim}(V_j)$ and S_j.

▶ **Observation 4.6.** $\bigcup_{j \in [n]} \mathtt{rim}(V_j) \subseteq \bigcup_{j \in [n]} S_j$

Proof. Consider a node $v \in \mathtt{rim}(V_j)$ for some $j \in [n]$. By definition, if $\mathrm{sec}(v)$ is empty, then $v \in S_j$. If $\mathrm{sec}(v)$ is not empty, then there exists some $j' \in [n]$ such that $v \in \mathtt{sec}(V_{j'})$, and therefore $v \in S_{j'}$. Overall, we get that $v \in \bigcup_{\ell \in [n]} S_\ell$. ◀

4.5 Labels and Verification

In this section, we describe the label assignment and verification process of our compiler for the case of MinDGPs. The description of the changes required to establish the same for MaxDGPs is deferred to the full version of this paper [8]. In both cases, we establish the proof size and correctness of our construction, thus proving Theorem 4.1.

Consider a canonical MinDGP $\Psi = \langle \Pi, f \rangle$ and an IO graph $G_{\mathtt{i},\mathtt{o}} \in \Pi$, where $\mathtt{o}$ is an optimal solution for G and $\mathtt{i}$. The prover uses the SLOCAL algorithm Part_OPT presented in Sec. 4.4 to compute the values $r(j)$ and subsets V_j, $\mathtt{sec}(V_j)$, $\mathtt{ext}(V_j) = V_j \cup \mathtt{sec}(V_j)$, and $S_j = \mathtt{sec}(V_j) \cup \{v \in \mathtt{rim}(V_j) \mid \mathrm{sec}(v) \text{ is empty}\}$ for all $j \in [n]$.

The goal of the prover is to provide proof of four properties satisfied by the given solution $\mathtt{o}$ and the outcome of Part_OPT. We refer to those properties as *feasibility*, *rim*, *growth*, and *optimality*. The four properties are defined as follows. The feasibility property states that $\mathtt{o}$

is a feasible solution for G and i, i.e., $G_{\mathsf{i},\mathsf{o}} \in \Pi$; the rim property states that for each $j \in [n]$ and node $v \in \mathtt{rim}(V_j)$, there exists $j' \in [n]$, such that $v \in S_{j'}$; the growth property states that $w(S_j, \mathsf{o}) \leq \epsilon \cdot w(\mathtt{inner}^2(V_j), \mathsf{o})$ for each $j \in [n]$; and the optimality property states that $w(\mathtt{inner}^2(V_j), \mathsf{o}) \leq \alpha \cdot w_{\min}(V_j)$ for each $j \in [n]$.

The prover provides its proof by means of a label assignment $L : V \to \{0,1\}^*$ that assigns each node v with a label $L(v) = \langle L_{\text{feas}}(v), L_{\text{rim}}(v), L_{\text{grw}}(v), L_{\text{opt}}(v)\rangle$. The label $L(v)$ is composed of the fields $L_{\text{feas}}(v)$, $L_{\text{rim}}(v)$, $L_{\text{grw}}(v)$, and $L_{\text{opt}}(v)$ that provide proof for the feasibility, rim, growth, and optimality properties, respectively.

The field $L_{\text{feas}}(\cdot)$ provides a proof for the feasibility property by setting $L_{\text{feas}}(v) = \mathsf{o}(v)$ for each node $v \in V$. Notice that since Π is an LCL, verifying o's feasibility is done by checking that $L_{\text{feas}}(v) = \mathsf{o}(v)$, and $p^{\Pi}_{\mathtt{prd}(\mathsf{i}(v))}(L_{\text{feas}}(v), L_{\text{feas}}(u_1), \ldots L_{\text{feas}}(u_{\deg_G(v)})) = \mathtt{True}$ at each node v with neighbors $u_1, \ldots, u_{\deg_G(v)}$.

The field $L_{\text{rim}}(\cdot)$ provides a proof for the rim property as follows. First, the sets V_j and S_j are encoded for all $1 \leq j \leq n$, where each of the sets is identified by $id(v_j)$. In addition, each node $v \in \mathtt{rim}(V_j)$ is assigned the minimal distance to a node $u \notin V_j$ (notice that by definition, these values are either 1 or 2). This allows the verifier to check that for each node $v \in \mathtt{rim}(V_j)$, there exists $j' \in [n]$ such that $v \in S_{j'}$, i.e., verify that the rim property is satisfied.

The field $L_{\text{grw}}(\cdot)$ provides a proof for the growth property simply by using a comparison scheme (as defined in Sec. 3) that compares between $w(S_j, \mathsf{o})$ and $\epsilon \cdot w(\mathtt{inner}^2(V_j), \mathsf{o})$. This comparison scheme is used concurrently for each $\mathtt{ext}(V_j) \neq \emptyset$, based on a shortest paths tree of $G(\mathtt{ext}(V_j))$ rooted at node v_j. Observe that by Lemma 4.4, this tree spans the nodes of $\mathtt{ext}(V_j)$ and has diameter $O(\log n)$.

The field $L_{\text{opt}}(\cdot)$ provides a proof for the optimality property as follows. First, for each $V_j \neq \emptyset$, the prover computes an assignment $g_j : V_j \to \{0, \ldots, k(\Pi, n)\}$, such that g_j respects Π and $w(V_j, g_j) = w_{\min}(V_j)$. The prover assigns each node $v \in V_j$ with the multiplicity $g_j(v)$ and proves that $w(\mathtt{inner}^2(V_j), \mathsf{o}) \leq w(V_j, g_j)$ by means of a comparison scheme based on a shortest paths (spanning) tree of $G(V_j)$ rooted at node v_j. Finally, the prover proves that $w(V_j, g_j) \leq \alpha \cdot w_{\min}(V_j)$ by means of an α-APLS for Ψ on the configured subgraph $G_{\mathsf{i},\mathsf{o}}(V_j)$. Notice that an α-APLS for Ψ is well-defined over the instance $G_{\mathsf{i},\mathsf{o}}(V_j)$ since Π is self-induced, and thus $G_{\mathsf{i}}(V_j) \in \mathcal{LEG}(\Pi)$.

Proof Size and Correctness. We observe that the label assignment produced by the prover can be computed by means of an SLOCAL algorithm with locality $O(\log n)$ and thus it can be simulated by a locally restricted prover. Moreover, for each node $v \in V$, the sub-labels $L_{\text{feas}}(v)$, $L_{\text{rim}}(v)$, and $L_{\text{grw}}(v)$ are of size $O(\log n)$; whereas $L_{\text{opt}}(v)$ is of size $\ell_{\Psi,\alpha} + O(\log n)$, where $\ell_{\Psi,\alpha}$ is the proof size of an α-APLS for Ψ. Overall, the proof size of this scheme is $\ell_{\Psi,\alpha} + O(\log n)$.

Regarding the correctness requirements, we start by showing the completeness requirement, i.e., we show that if o is an optimal solution for G and i, then the verifier accepts $G_{\mathsf{i},\mathsf{o}}$. To that end, it is sufficient to show that all four aforementioned properties are satisfied. The feasibility property holds since by definition, o is a feasible solution for G and i; the rim property follows directly from Observation 4.6; the growth property holds by the construction of the clusters V_j; and the optimality property follows from Lemma 4.2. We note that as established in Lemma 4.2, the optimality property is satisfied by o with parameter $\alpha = 1$. However, providing proof for this stronger property might be costly in terms of proof size. Thus, to obtain a small proof size, we settle for an approximated version.

As for the soundness requirement, consider an IO graph $G_{\mathsf{i},\mathsf{o}}$ such that the verifier accepts $G_{\mathsf{i},\mathsf{o}}$. This means that all four properties hold for $G_{\mathsf{i},\mathsf{o}}$. First, observe that by the feasibility property, it holds that $G_{\mathsf{i},\mathsf{o}} \in \Pi$. Let $V_1^L, \ldots V_k^L$ and $S_1^L, \ldots S_k^L$ be the subsets V_j and S_j encoded by the prover in the field $L_{\mathrm{rim}}(\cdot)$. By the rim property, it holds that $\bigcup_{j \in [k]} \mathtt{rim}(V_j^L) \subseteq \bigcup_{j \in [k]} S_j^L$. From the growth property it follows that $\epsilon \cdot w(\bigcup_{j \in [k]} \mathtt{inner}^2(V_j^L), \mathsf{o}) \geq w(\bigcup_{j \in [k]} S_j^L, \mathsf{o}) \geq w(\bigcup_{j \in [k]} \mathtt{rim}(V_j^L), \mathsf{o})$. Let $\mathsf{o}^* : V \to \{0, \ldots, k(\Pi, n)\}$ be an optimal solution for G and i. We observe that for any $U \subseteq V$, the assignment of o^* on the nodes of U must respect Π, and therefore $w(U, \mathsf{o}^*) \geq w_{\min}(U)$. The optimality property combined with the last observation implies that $w(\bigcup_{j \in [k]} \mathtt{inner}^2(V_j^L), \mathsf{o}) \leq \alpha(w_{\min}(V_1^L) + \cdots + w_{\min}(V_k^L)) \leq \alpha(w(V_1^L, \mathsf{o}^*) + \cdots + w(V_k^L, \mathsf{o}^*)) = \alpha \cdot w(V, \mathsf{o}^*) = \alpha \cdot f(G_{\mathsf{i},\mathsf{o}^*})$. Combining this inequality with the rim and growth properties implies that $f(G_{\mathsf{i},\mathsf{o}}) = w(V, \mathsf{o}) = w(\bigcup_{j \in [k]} \mathtt{inner}^2(V_j^L), \mathsf{o}) + w(\bigcup_{j \in [k]} \mathtt{rim}(V_j^L), \mathsf{o}) \leq (1 + \epsilon) \cdot w(\bigcup_{j \in [k]} \mathtt{inner}^2(V_j^L), \mathsf{o}) \leq \alpha \cdot (1 + \epsilon) \cdot f(G_{\mathsf{i},\mathsf{o}^*})$, thus establishing the soundness requirement.

5 Compiler for CGFs

In this section, we present our generic compiler for CGFs. It is divided into three subsections as follows. First, in Sec. 5.1 we characterize the CGFs that are suited for our compiler, namely SU-closed CGFs. Following that, Sec. 5.3 and 5.4 are dedicated to the compiler construction. More formally, these sections constructively prove the following theorem.

▶ **Theorem 5.1.** *Let Φ be an SU-closed CGF that admits a PLS with a proof size of ℓ_Φ. For any constant $\delta > 0$, there exists a locally restricted δ-TPLS for Φ with a proof size of $\ell_\Phi + O(\log n)$.*

For convenience, the compiler construction is divided between Sec. 5.3, in which we present an SLOCAL partition algorithm (that plays a similar role to the one presented in the OptDGP compiler), and Sec. 5.4, in which we describe the label assignment and verification process. In Sec. 5.2, we provide a high-level overview of the SLOCAL algorithm and how it is used in the label assignment and verification process.

5.1 SU-Closed CGFs

A CGF Φ is said to be *closed under node-induced subgraphs* if for every configured graph $G_s \in \Phi$ and node subset $U \subseteq V$, it holds that $G_s(U) \in \Phi$. We say that two configured graphs $G_s = \langle G = (V, E), s \rangle$ and $G'_{s'} = \langle G' = (V', E'), s' \rangle$ are *disjoint* if $V \cap V' = \emptyset$. We define the *disjoint union* between two disjoint configured graphs $G_s = \langle G = (V, E), s \rangle$ and $G'_{s'} = \langle G' = (V', E'), s' \rangle$ as the configured graph $\tilde{G}_{\tilde{s}} = \langle \tilde{G}, \tilde{s} \rangle$ consisting of the graph $\tilde{G} = (V \dot{\cup} V', E \dot{\cup} E')$ and the configuration function $\tilde{s} : V \dot{\cup} V' \to \{0, 1\}^*$ that assigns the local configuration $\tilde{s}(v) = s(v)$ to any node $v \in V$; and $\tilde{s}(v) = s'(v)$ to any node $v \in V'$. We say that a CGF Φ is *closed under disjoint union* if for any two disjoint configured graphs $G_s, G'_{s'} \in \Phi$ with disjoint union $\tilde{G}_{\tilde{s}}$, it holds that $\tilde{G}_{\tilde{s}} \in \Phi$. We refer to a CGF Φ as *SU-closed* if it is closed under node-induced subgraphs and under disjoint union.

5.2 Overview

In Sec. 5.3, we present an SLOCAL algorithm called `Part_CGF` that partitions the nodes of a given configured graph G_s into clusters. Before formally describing the algorithm in Sec. 5.3, let us provide some intuition by presenting the high-level idea of the partition and how it is used to design the label assignment and verification process for an SU-closed CGF Φ.

Given a configured graph $G_s \in \Phi$, we use a ball growing argument to obtain a partition of the nodes into clusters such that: (1) the subgraph induced by each cluster is of logarithmic diameter; and (2) the number of crossing edges between clusters is a δ-fraction of the number of edges in the clusters.

In order to allow the prover to provide a proof for the second property by local means, during `Part_CGF`, some nodes may be assigned a *secondary affiliation* to an adjacent cluster. The idea is that for each cluster V_j, the number of crossing edges to nodes with secondary affiliation to V_j is a δ-fraction of the number of edges within V_j.

5.3 Partition Algorithm

Algorithm's Description. We now provide a formal description of the `Part_CGF` algorithm. Consider an SU-closed CGF Φ and a configured graph $G_s \in \Phi$. The algorithm partitions the nodes of G into (possibly empty) clusters $V = V_1 \dot{\cup} \ldots \dot{\cup} V_n$. As usual in the SLOCAL model, the nodes are processed sequentially in n iterations based on an arbitrary order $v_1, \ldots, v_n$ on the nodes, where node v_j is processed in the j-th iteration.

Throughout the execution of `Part_CGF`, each node $v \in V$ maintains two fields referred to as $\text{cluster}(v)$ and $\text{sec}(v)$. The field $\text{cluster}(v)$ is initially empty and its role is to identify v's cluster, where each cluster V_j is identified by the id of node v_j which is processed in the j-th iteration, i.e., $V_j = \{v \mid \text{cluster}(v) = id(v_j)\}$. The field $\text{sec}(v)$ is initially empty and its role is to identify v's *secondary affiliation* to a cluster if such affiliation exists (otherwise it remains empty throughout the algorithm).

The j-th iteration of `Part_CGF` is executed as follows. Let G_j to be the subgraph induced on G by $V \setminus (V_1 \cup \cdots \cup V_{j-1})$. If $v_j \in V_1 \cup \cdots \cup V_{j-1}$, then we define $V_j = \emptyset$ and finish the iteration; so, assume that v_j is a node in G_j. For an integer $r \in \mathbb{Z}_{\geq 0}$, let D_j^r be the set of nodes at distance exactly r from v_j in G_j and let $B_j^r = \bigcup_{r'=0}^{r} D_j^{r'}$. Let E_j^r be the set of edges in the subgraph $G(B_j^r)$ and let C_j^r be the set of edges $(u, v) \in E$, such that $u \in B_j^r$ and $v \notin B_j^r$. Define $r(j)$ to be the smallest integer that satisfies $|C_j^{r(j)}| \leq \delta \cdot |E_j^{r(j)}|$. The j-th iteration is completed by setting $\text{cluster}(v) = id(v_j)$ for each node $v \in B_j^{r(j)}$; and $\text{sec}(v) = id(v_j)$ for each node $v \in D_j^{r(j)+1}$.

Algorithm's Properties. The following lemma establishes an upper bound on the diameter of each subgraph $G(V_j)$.

▶ **Lemma 5.2.** *The diameter of subgraph $G(V_j)$ is $O(\log n)$ for each $j \in [n]$.*

Proof. Suppose that $V_j \neq \emptyset$ (as the lemma is trivial otherwise). To show that the diameter of $G(V_j)$ is $O(\log n)$ it is sufficient to show that $r(j) = O(\log n)$. We use a ball growing argument. Note that for any integer r, it holds that $|E_j^r| \geq |E_j^{r-1}| + |C_j^{r-1}|$. Thus, for every $r' < r(j) + 1$, we have

$$|E_j^{r(j)+1}| \geq |E_j^{r'}| > (1 + \delta) \cdot |E_j^{r'-1}| > (1 + \delta)^2 \cdot |E_j^{r'-2}| > \ldots$$

and since $n^2 > m \geq |E_j^{r(j)+1}|$, we get that $r(j) = O((1/\delta) \log n) = O(\log n)$. ◀

A simple observation derived from Lemma 5.2 is that `Part_CGF` has locality $O(\log n)$. This observation combined with the results of [13, 27] lead to the following corollary.

▶ **Corollary 5.3.** *The algorithm `Part_CGF` can be simulated by a LOCAL algorithm with polylogarithmic round-complexity.*

5.4 Labels and Verification

Consider an SU-closed CGF Φ and a configured graph $G_s \in \Phi$. The prover uses the SLOCAL algorithm `Part_CGF` presented in Sec. 5.3 to compute the values $r(j)$ for all $1 \leq j \leq n$, and the fields cluster(v), sec(v). The goal of the prover is to provide proof of two properties satisfied by the given configured graph G_s and the outcome of `Part_CGF`. We refer to those properties as *secondary clusters*, *crossing edges*, and *inclusion*. To that end, the prover produces a label assignment $L : V \to \{0,1\}^*$ that assigns each node v with a label $L(v) = \langle L_{\text{sec}}(v), L_{\text{cross}}(v), L_{\text{inc}}(v)\rangle$. The label $L(v)$ is composed of the fields $L_{\text{sec}}(v)$, $L_{\text{cross}}(v)$, and $L_{\text{inc}}(v)$, that provide proof for the secondary clusters, crossing edges and inclusion properties, respectively.

The secondary clusters property states that sec(v) is not empty for every node v that has a neighbor belonging to a different cluster. To that end, the sub-label $L_{\text{sec}}(v)$ assigns the values cluster(v) and sec(v) to each node $v \in V$. Observe that this information is sufficient for the verifier to verify the secondary clusters property.

For all $j \in [n]$, let $\texttt{sec}(V_j) = \{v \mid \text{sec}(v) = id(v_j)\}$ be the set of nodes whose secondary affiliation is to V_j by the end of the `Part_CGF` algorithm, and let $F_j = \{(u, v) \in E \mid u \in V_j, v \in \texttt{sec}(V_j)\}$ denote the set of edges with one endpoint in V_j and the other endpoint in $\texttt{sec}(V_j)$. The crossing edges property states that each cluster V_j satisfies $|F_j| \leq \delta \cdot |E_j^{r(j)}|$. The field $L_{\text{cross}}(\cdot)$ serves the crossing edges property by means of a comparison scheme between $|F_j|$ and $\delta \cdot |E_j^{r(j)}|$. This comparison scheme is based on a shortest paths (spanning) tree rooted at node v_j for each cluster $V_j \neq \emptyset$. Notice that each node $v \in V_j$ knows its incident edges from $|F_j|$ based on the $L_{\text{sec}}(\cdot)$ field of its neighbors.

The inclusion property states that $G_s(V_j) \in \Phi$ for all $V_j \neq \emptyset$. To that end, the prover uses the field $L_{\text{inc}}(\cdot)$ to encode a PLS for Φ concurrently on all subgraphs $G(V_j)$.

Proof Size and Correctness. We observe that the label assignment produced by the prover can be computed by means of an SLOCAL algorithm with locality $O(\log n)$ and thus it can be simulated by a locally restricted prover. Moreover, the sub-labels $L_{\text{sec}}(v)$ and $L_{\text{cross}}(v)$ are of size $O(\log n)$; and $L_{\text{inc}}(v)$ is of size ℓ_Φ, where ℓ_Φ is the proof size of a PLS for Φ. Overall, the proof size of this scheme is $\ell_\Phi + O(\log n)$.

We now show that the correctness requirements are satisfied. We start with completeness, i.e., showing that if $G_s \in \Phi$, then the verifier accepts G_s. Observe that the secondary clusters and crossing edges properties are satisfied by construction of `Part_CGF`. In addition, the inclusion property follows from the fact that Φ is closed under node-induced subgraphs.

As for the soundness requirement, consider a configured graph G_s such that the verifier accepts G_s. Let $V_1^L, \dots V_k^L$ be the clusters encoded in the field $L_{\text{sec}}(\cdot)$. For every $j \in [k]$, let E_j^L denote the edge set of subgraph $G(V_j^L)$, let $\texttt{sec}_j^L$ be the set of nodes for which the field $L_{\text{sec}}(\cdot)$ encodes a secondary affiliation to V_j^L, and let F_j^L be the set of edges with one endpoint in V_j^L and one in $\texttt{sec}_j^L$. The inclusion property guarantees that $G_s(V_j^L) \in \Phi$ for each $j \in [k]$. Let $G'_{s'}$ be the disjoint union of $G_s(V_1^L), \dots, G_s(V_k^L)$. Since Φ is closed under disjoint union, we get that $G'_{s'} \in \Phi$. By the secondary clusters property, $G'_{s'}$ is the configured subgraph obtained from G_s by removing the set $F_1^L \cup \cdots \cup F_k^L$ of edges. The crossing edges property implies that $|F_1^L \cup \cdots \cup F_k^L| = \sum_{j\in[k]} |F_j^L| \leq \delta \cdot \sum_{j\in[k]} |E_j^L| \leq \delta m$. Thus, G_s is not δ-far from belonging to Φ, i.e., $G_s \notin \mathcal{F}_N$.

In conclusion, this scheme describes a correct locally restricted δ-TPLS for SU-closed CGFs with a proof size of $\ell_\Phi + O(\log n)$, thus proving Theorem 5.1.

References

1. Noga Alon, Tali Kaufman, Michael Krivelevich, and Dana Ron. Testing triangle-freeness in general graphs. *SIAM J. Discret. Math.*, 2008.
2. B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 268–277, 1991.
3. Baruch Awerbuch, Andrew V. Goldberg, Michael Luby, and Serge A. Plotkin. Network decomposition and locality in distributed computation. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October – 1 November 1989*, pages 364–369. IEEE Computer Society, 1989.
4. Nir Bacrach, Keren Censor-Hillel, Michal Dory, Yuval Efron, Dean Leitersdorf, and Ami Paz. Hardness of distributed optimization. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019.
5. Lélia Blin, Pierre Fraigniaud, and Boaz Patt-Shamir. On proof-labeling schemes versus silent self-stabilizing algorithms. In *Stabilization, Safety, and Security of Distributed Systems*, pages 18–32, 2014.
6. Keren Censor-Hillel, Ami Paz, and Mor Perry. Approximate proof-labeling schemes. *Theor. Comput. Sci.*, 811:112–124, 2020.
7. Yuval Emek and Yuval Gil. Twenty-two new approximate proof labeling schemes. In *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, 2020.
8. Yuval Emek, Yuval Gil, and Shay Kutten. Locally restricted proof labeling schemes (full version), 2022. `arXiv:2208.08718`.
9. Laurent Feuilloley and Pierre Fraigniaud. Error-sensitive proof-labeling schemes. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 16:1–16:15, 2017.
10. Laurent Feuilloley, Pierre Fraigniaud, Juho Hirvonen, Ami Paz, and Mor Perry. Redundancy in distributed proofs. In *32nd International Symposium on Distributed Computing, DISC*, volume 121 of *LIPIcs*, pages 24:1–24:18, 2018.
11. Laurent Feuilloley, Pierre Fraigniaud, Pedro Montealegre, Ivan Rapaport, Éric Rémila, and Ioan Todinca. Compact distributed certification of planar graphs. *Algorithmica*, 83(7):2215–2244, 2021.
12. Pierre Fraigniaud, Amos Korman, and David Peleg. Local distributed decision. In *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS*, pages 708–717, 2011.
13. Mohsen Ghaffari, Fabian Kuhn, and Yannic Maus. On the complexity of local distributed graph problems. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 784–797. ACM, 2017.
14. Oded Goldreich, Shafi Goldwasser, and Dana Ron. Property testing and its connection to learning and approximation. *J. ACM*, 45(4):653–750, 1998.
15. Shafi Goldwasser, Guy N. Rothblum, and Yael Tauman Kalai. Delegating computation: Interactive proofs for muggles. *Electron. Colloquium Comput. Complex.*, page 108, 2017.
16. Mika Göös and Jukka Suomela. Locally checkable proofs in distributed computing. *THEORY OF COMPUTING*, 12:1–33, 2016.
17. Gillat Kol, Rotem Oshman, and Raghuvansh R. Saxena. Interactive distributed proofs. In *PODC 2018 - Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 255–264, July 2018.
18. Amos Korman and Shay Kutten. Distributed verification of minimum spanning trees. *Distributed Comput.*, 20(4):253–266, 2007.
19. Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Comput.*, 22(4):215–233, 2010.
20. E. Kushilevitz and N. Nisan. *Communication Complexity.* Cambridge University Press, 2006.

21 Nathan Linial. Distributive graph algorithms-global solutions from local data. In *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*, pages 331–335. IEEE Computer Society, 1987.

22 Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992.

23 Moni Naor and Larry J. Stockmeyer. What can be computed locally? *SIAM J. Comput.*, 24(6):1259–1277, 1995.

24 Boaz Patt-Shamir and Mor Perry. Proof-labeling schemes: Broadcast, unicast and in between. In *Stabilization, Safety, and Security of Distributed Systems - 19th International Symposium, SSS*, volume 10616, pages 1–17. Springer, 2017.

25 David Peleg. *Distributed Computing: A Locality-Sensitive Approach.* Society for Industrial and Applied Mathematics, USA, 2000.

26 Omer Reingold, Guy N. Rothblum, and Ron D. Rothblum. Constant-round interactive proofs for delegating computation. *SIAM J. Comput.*, 50(3), 2021.

27 Václav Rozhon and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 350–363. ACM, 2020.

A Bounds for Concrete OptDGPs and CGFs

A.1 OptDGPs

In this section, we show how the compiler presented in Section 4 can be used in the design of locally restricted APLSs for some classical OptDGPs that fit the canonical structure. In Sections A.1.1, A.1.2, and A.1.3, we present locally restricted APLSs with a logarithmic proof size for the problems of minimum weight vertex cover, maximum independent set, and minimum weight dominating set, respectively. Then, in Section A.1.4, we present a locally restricted $(1+\epsilon)$-APLS that applies to any canonical OptDGP.

A.1.1 Minimum Weight Vertex Cover

Consider a graph $G = (V, E)$ associated with a node-weight function $w : V \to \{1, \dots, n^{O(1)}\}$ and let $C \subseteq E$ be a set of *constrained* edges. A *vertex cover* of C is a subset $U \subseteq V$ of nodes such that every edge $e \in C$ has at least one endpoint in U. A *minimum weight vertex cover (MWVC)* of C is a vertex cover U of C that minimizes $w(U) = \sum_{u \in U} w(u)$.

Observe that MWVC is a covering OptDGP. Moreover, vertex cover is self-induced (notice that this is in contrast to the common case of vertex cover where all edges are constrained). Thus, MWVC is canonical. We aim to use our compiler to construct a locally restricted $2(1+\epsilon)$-APLS for MWVC. To that end, we establish the following lemma.

▶ **Lemma A.1.** *There exists a* 2*-APLS for MWVC with a proof size of* $O(\log n)$*.*

Proof. As presented in [7], there exists a 2-APLS for the instance of MWVC where all edges are constrained and the graph is connected. We can obtain a 2-APLS for MWVC in the more general case where a subset $C \subseteq E$ of edges are constrained simply by applying the 2-APLS from [7] on the connected subgraphs induced by the constrained edge set C. The proof size of this scheme is $O(\log n + \log W)$, where W is an upper bound on the node-weights. Since in our case $W = n^{O(1)}$, it follows that the proof size is $O(\log n)$. ◀

Plugging the 2-APLS obtained in Lemma A.1 into our compiler leads to the following corollary.

▶ **Corollary A.2.** *For any constant $\epsilon > 0$, there exists a locally restricted $(2(1+\epsilon))$-APLS for MWVC with a proof size of $O(\log n)$.*

We now consider the unweighted version, simply referred to as *minimum vertex cover (MVC)*, on graphs with large odd-girth (where the odd-girth of a graph is defined to be the shortest odd cycle). As the following theorem shows, this case allows for an improved approximation ratio.

▶ **Theorem A.3.** *For any constant $\epsilon > 0$, there exists a locally restricted $(1+\epsilon)$-APLS for MVC on graphs of odd-girth $\omega(\log n)$ with a proof size of $O(\log n)$.*

Proof. Recall that our compiler first partitions the nodes into clusters of diameter $O(\log n)$, and then proceeds to apply an α-APLS concurrently on the subgraph induced by each cluster. We observe that each of these subgraphs created by the partition is bipartite since the odd-girth of the graph is $\omega(\log n)$. Thus, it is sufficient to show that there exists a PLS for MVC on bipartite graphs with a proof size of $O(\log n)$.

The well known König's theorem states that in bipartite graphs the size of minimum vertex cover is equal to the size of maximum matching. This allows for a PLS for MVC in bipartite graphs with a proof size of $O(\log n)$ constructed as follows. The prover simply encodes a maximum matching on the graph along with a proof that the size of this matching is equal to the size of the given vertex cover (e.g., by means of a comparison scheme). ◀

A.1.2 Maximum Independent Set

Consider a graph $G = (V, E)$ and let $C \subseteq E$ be a set of *constrained* edges. An *independent set* of C is a subset $I \subseteq V$ of nodes, such that each node $v \in I$ is incident on an edge in C and every edge $e \in C$ has at most one endpoint in I. A *maximum independent set (MaxIS)* of C is an independent set I of C that maximizes $|I|$.

Observe that MaxIS is a packing OptDGP. Moreover, independent set is self-induced (notice that this is in contrast to the common case of independent set where all edges are constrained). Thus, MaxIS is canonical.

Let us denote by $\Delta = \max_{v \in V}\{\deg(v)\}$ the largest degree in graph $G = (V, E)$. In the following lemma, we present a simple Δ-APLS for the MaxIS problem.

▶ **Lemma A.4.** *There exists a Δ-APLS for MaxIS with a proof size of $O(\log n)$.*

Proof. We use the fact that the ratio between the size of a maximum independent set and the size of a maximal independent set (i.e., an independent set that is not a subset of any other independent set) is at most Δ. The Δ-APLS construction is simple. The prover encodes a maximal independent set along with the value of Δ and a proof that its size is at most a multiplicative factor of Δ away from the given independent set. ◀

Plugging the Δ-APLS from Lemma A.4 into our compiler leads to the following corollary.

▶ **Corollary A.5.** *For any constant $\epsilon > 0$, there exists a locally restricted $(\Delta(1+\epsilon))$-APLS for MaxIS with a proof size of $O(\log n)$.*

Similarly to the MVC problem, restricting MaxIS to families of graphs with odd-girth $\omega(\log n)$ allows for a better approximation ratio, as established by the following theorem.

▶ **Theorem A.6.** *For any constant $\epsilon > 0$, there exists a locally restricted $(1+\epsilon)$-APLS for MaxIS on graphs of odd-girth $\omega(\log n)$ with a proof size of $O(\log n)$.*

Proof. Similarly to the proof of Theorem A.3, it is sufficient to show that there exists a PLS for MaxIS on bipartite graphs with a proof size of $O(\log n)$. To that end, we can use the fact that in bipartite graphs, the size of MaxIS is equal to the size of a minimum edge cover. We can now construct a PLS where the prover encodes a minimum edge cover of the graph, along with a proof that it is equal in size to the given independent set. ◀

A.1.3 Minimum Weight Dominating Set

Consider a graph $G = (V, E)$ associated with a node-weight function $w : V \to \{1, \dots n^{O(1)}\}$ and let $C \subseteq V$ be a subset of *constrained* nodes. A *dominating set* of C is a subset $D \subseteq V$ of nodes, such that $D \cap (v \cup N(v)) \neq \emptyset$ for each constrained node $v \in C$. A *minimum weight dominating set (MWDS)* of C is a dominating set D of C that minimizes $\sum_{u \in D} w(u)$.

Observe that MWDS is a covering OptDGP. Moreover, dominating set is self-induced (notice that this is in contrast to the common case of dominating set where all nodes are constrained). Thus, we get that MWDS is canonical. We aim to use our compiler to construct a locally restricted $O(\log n)$-APLS for MWDS. To that end, we establish the following lemma.

▶ **Lemma A.7.** *There exists an $O(\log n)$-APLS for MWDS with a proof size of $O(\log n)$.*

Proof. An $O(\log n)$-APLS for the instance of MWDS where all nodes are constrained and is presented in [7]. The idea behind that $O(\log n)$-APLS is that the prover provides a feasible solution to the dual LP, such that the objective value of this dual solution is at most a multiplicative factor of $O(\log n)$ from the given dominating set. We argue that this technique can be applied to obtain $O(\log n)$-APLS for MWDS (in its more generalized version described above). This follows from the fact that the gap between an optimal MWDS solution and an optimal dual solution remains $O(\log n)$ (since MWDS is an instance of set cover). ◀

Plugging the $O(\log n)$-APLS obtained in Lemma A.7 into our compiler leads to the following corollary.

▶ **Corollary A.8.** *There exists a locally restricted $O(\log n)$-APLS for MWDS with a proof size of $O(\log n)$.*

A.1.4 Generic Locally Restricted $(1 + \epsilon)$-APLS for Canonical OptDGPs

We establish a generic upper bound that applies to any canonical OptDGP.

▶ **Theorem A.9.** *Consider a canonical OptDGP Ψ. For any constant $\epsilon > 0$, there exists a locally restricted $(1 + \epsilon)$-APLS for Ψ with a proof size of $O(n^2)$.*

Proof. As stated in [19, Theorem 3.2], any decidable property admits a PLS. The idea behind this universal PLS is that the prover can assign each node v with a label $L(v)$ that encodes the entire configured graph G_s. In response, the verifier at node v verifies that v's neighbors agree on the structure of the configured graph encoded in $L(v)$, and that v's local neighborhood is consistent with the one encoded in $L(v)$. Following that, the verifier can evaluate whether G_s is a yes-instance or not.

Observe that applying the universal PLS described above for a canonical OptDGP requires a proof size of $O(n^2)$. We can now plug this PLS construction into our compiler to obtain the desired locally restricted $(1 + \epsilon)$-APLS for Ψ. ◀

A.2 CGFs

In this section, we show how the compiler presented in Section 5 can be combined with known PLS constructions to obtain locally restricted δ-TPLSs for various well-known SU-closed CGFs.

A.2.1 Planarity

A graph $G = (V, E)$ is called *planar* if it can be embedded in the plane. The following lemma has been established by Feuilloley et al. [11].

▶ **Lemma A.10.** *There exists a PLS for planarity with a proof size of $O(\log n)$.*

Observe that planar graphs are SU-closed. Thus, plugging the PLS for planarity into our compiler implies the following corollary.

▶ **Corollary A.11.** *For any constant $\delta > 0$, there exists a locally restricted δ-TPLS for planarity with a proof size of $O(\log n)$.*

A.2.2 Bounded Arboricity

The *arboricity* of a graph $G = (V, E)$ is the minimum number k for which there exists an edge partition $E = E_1 \dot\cup \ldots \dot\cup E_k$ such that $G_i = (V, E_i)$ is a forest for each $i \in [k]$. Let $arb(G)$ denote the arboricity of graph G. We say that graph G is of bounded arboricity if $arb(G) = O(1)$.

▶ **Lemma A.12.** *There exists a PLS for bounded arboricity with a proof size of $O(\log n)$.*

Proof. As established in [19], there exists a PLS for forests with a proof size of $O(\log n)$. A PLS for bounded can be implemented by using the PLS construction for forests concurrently on $arb(G)$ edge-induced subgraphs of G. ◀

Observe that bounded arboricity is SU-closed. Thus, plugging the PLS for bounded arboricity into our compiler implies the following corollary.

▶ **Corollary A.13.** *For any constant $\delta > 0$, there exists a locally restricted δ-TPLS for bounded arboricity with a proof size of $O(\log n)$.*

A.2.3 k-Colorability

For a positive integer k, we say that a graph $G = (V, E)$ is *k-colorable* if there exists a proper k-coloring of its nodes. Observe that k-colorability admits a (simple) PLS with proof size $O(\log k)$ and that it is SU-closed. Thus, combined with our compiler, we get the following theorem.

▶ **Theorem A.14.** *For any constant $\delta > 0$, there exists a locally restricted δ-TPLS for k-colorability with a proof size of $O(\log n)$.*

A.2.4 Forests and DAGs

The following two lemmas have been established by Korman et al. [19].

▶ **Lemma A.15.** *There exists a PLS for forests with a proof size of $O(\log n)$.*

▶ **Lemma A.16.** *There exists a PLS for directed acyclic graphs (DAGs) with a proof size of* $O(\log n)$*.*

Observe that both forests and DAGs are SU-closed. Thus, plugging the PLSs for forests and DAGs into our compiler implies the following corollaries.

▶ **Corollary A.17.** *For any constant* $\delta > 0$*, there exists a locally restricted* δ*-TPLS for forests with a proof size of* $O(\log n)$*.*

▶ **Corollary A.18.** *For any constant* $\delta > 0$*, there exists a locally restricted* δ*-TPLS for directed acyclic graphs with a proof size of* $O(\log n)$*.*

B Impossibilities of Locally Restricted GPLS

In this section, we establish some inherent limitations of locally restricted GPLSs based on the following observation.

▶ **Observation B.1.** *If there exists a locally restricted GPLS over* $\mathcal{U}$ *with yes-family* $\mathcal{F}_Y$ *and no-family* $\mathcal{F}_N$*, then there exists a LOCAL algorithm with a* $\log^{O(1)}(n)$ *round-complexity that given a configured graph* $G_s \in \mathcal{U}$*, decides if* $G_s \in \mathcal{F}_Y$ *(in which case all nodes return* `True`*); or* $G_s \in \mathcal{F}_N$ *(in which case at least one node returns* `False`*).*

Proof. Given a configured graph $G_s \in \mathcal{U}$, we obtain a LOCAL algorithm by first simulating the locally restricted prover on G_s (using a polylogarithmic number of rounds), and then simulating the verifier (using 1 round). By the correctness requirements of a GPLS, the outcome of this algorithm is that all nodes return `True` if $G_s \in \mathcal{F}_Y$; whereas at least one node returns `False` if $G_s \in \mathcal{F}_N$. ◀

The observation above implies that it is impossible to construct a locally restricted GPLS for verification tasks that require $\omega(poly \log n)$ rounds in the LOCAL model. Notice that this impossibility applies to a large class of verification tasks associated with OptDGPs and CGFs. For example, using a simple indistinguishability argument, one can show that there is no locally restricted PLS for forests (i.e., a PLS deciding if a given graph is a forest). Similar arguments can be applied to exclude a locally restricted PLS for most of the OptDGPs and CGFs considered in Section A.

C Additional Related Work

The PLS model was introduced by Korman, Kutten, and Peleg in [19] and studied extensively since then, see, e.g., [11, 18, 5, 9, 24, 10]. Research in this field include [18], where a PLS for minimum spanning tree is shown to have a proof size of $O(\log n \log W)$, where W is the largest edge-weight, and [11], where a PLS for planarity is shown to have a proof size of $O(\log n)$.

In parallel, several researchers explored the limitations of the PLS model, often relying on known lower bounds from nondeterministic communication complexity [20]. Lower bounds of $\Omega(n^2)$ and $\Omega(n^2/\log n)$ are established in [16] with regards to the proof size of any PLS for graph symmetry and non 3-colorability, respectively. A similar technique was used by the authors of [4] to show that many classic optimization problems require a proof size of $\tilde{\Omega}(n^2)$.

The lower bounds on the proof size of PLSs for some optimization problems have motivated the authors of [6] to introduce the APLS notion, further studied recently in [7]. Optimization problems considered in the context of APLS include maximum weight matching, which was

shown in [6] to admit a 2-APLS with a proof size of $O(\log W)$, and minimum weight vertex cover, which was shown in [7] to admit a 2-APLS with a proof size of $O(\log n + \log W)$, where in both cases W refers to the largest weight value.

In the current paper, we also introduce the TPLS model which is suited for properties that are not formulated as optimization problems. This model is based on the notion of property testing [14]. More specifically, the TPLS model is formulated using the distance measure between graphs defined in [1].

Our focus in this paper is on locally restricted APLSs and TPLSs, restricting the prover to a LOCAL algorithm with a polylogarithmic number of rounds. Interest in the power of deterministic LOCAL algorithms with polylogarithmic round-complexity was initiated by Linial's seminal work [21, 22]. One particular problem that raised a lot of interest in this context is the network decomposition problem introduced by Awerbuch et al. in [3]. In a recent breakthrough [27], Ghaffari and Rozhon presented a deterministic algorithm with polylogarithmic round-complexity for the network decomposition problem. As established in [13], a consequence of this result is that any SLOCAL algorithm with $\log^{O(1)} n$ locality can be simulated by a LOCAL algorithm with $\log^{O(1)} n$ rounds. This simulation technique is used in the construction of our compilers in Sections 4 and 5.

Distributed Construction of Lightweight Spanners for Unit Ball Graphs

David Eppstein
Department of Computer Science, University of California, Irvine, CA, USA

Hadi Khodabandeh
Department of Computer Science, University of California, Irvine, CA, USA

Abstract

Resolving an open question from 2006 [14], we prove the existence of light-weight bounded-degree spanners for unit ball graphs in the metrics of bounded doubling dimension, and we design a simple $\mathcal{O}(\log^* n)$-round distributed algorithm in the LOCAL model of computation, that given a unit ball graph G with n vertices and a positive constant $\epsilon < 1$ finds a $(1+\epsilon)$-spanner with constant bounds on its maximum degree and its lightness using only 2-hop neighborhood information. This immediately improves the best prior lightness bound, the algorithm of Damian, Pandit, and Pemmaraju [13], which runs in $\mathcal{O}(\log^* n)$ rounds in the LOCAL model, but has a $\mathcal{O}(\log \Delta)$ bound on its lightness, where Δ is the ratio of the length of the longest edge to the length of the shortest edge in the unit ball graph. Next, we adjust our algorithm to work in the CONGEST model, without changing its round complexity, hence proposing the first spanner construction for unit ball graphs in the CONGEST model of computation. We further study the problem in the two dimensional Euclidean plane and we provide a construction with similar properties that has a constant average number of edge intersections per node. Lastly, we provide experimental results that confirm our theoretical bounds, and show an efficient performance from our distributed algorithm compared to the best known centralized construction.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Sparsification and spanners; Networks → Network control algorithms

Keywords and phrases spanners, doubling metrics, distributed, topology control

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.21

Related Version *Full Version*: https://arxiv.org/abs/2106.15234 [24]

1 Introduction

Given a collection of points V in a metric space with doubling dimension d, the weighted *unit ball graph (UBG)* on V is defined as a weighted graph $G(V, E)$ where two points $u, v \in V$ are connected if and only if their metric distance $\|uv\| \leq 1$. The weight of the edge uv of the UBG is $\|uv\|$ if the edge exists. Unit ball graphs in the Euclidean plane are called unit disk graphs (UDGs) and are frequently used to model ad-hoc wireless communication networks, where every node in the network has an effective communication range R, and two nodes are able to communicate if they are within a distance R of each other.

Spanners are sub-graphs of the input graph whose pair-wise distances approximate distances in the input graphs, while having fewer edges than complete graphs. Given a weighted graph G, a t-spanner on G can be defined as a graph S that has $V(G)$ as its set of vertices, while $E(S) \subseteq E(G)$ and the following inequality is satisfied for every pair of vertices $u, v \in V(G)$:

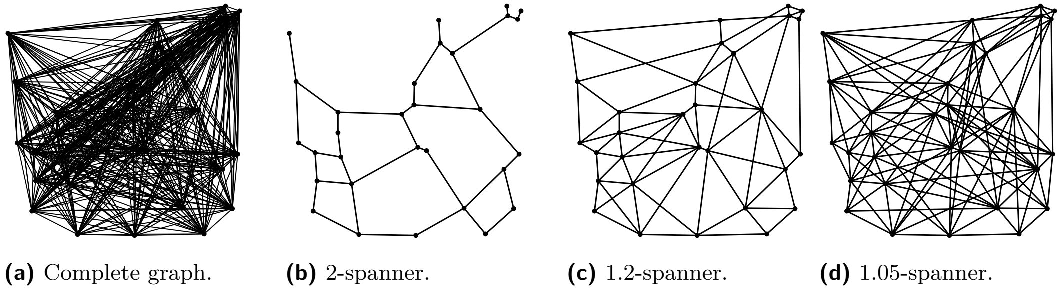
$$\mathrm{dist}_S(u, v) \leq t \cdot \mathrm{dist}_G(u, v)$$

where $\mathrm{dist}_S(u, v)$ (or $\mathrm{dist}_G(u, v)$) is the length of the shortest path between u and v using the edges in S (or G, respectively). We call this inequality the *bounded stretch property*. Because of this inequality, t-spanners provide a t-approximation for the pairwise distances

© David Eppstein and Hadi Khodabandeh;
licensed under Creative Commons License CC-BY 4.0

between the vertices in G. The parameter $t > 1$ is called the *stretch factor* or *spanning ratio* of the spanner and determines how accurate the approximate distances are; spanners having smaller stretch factors are more accurate.

Spanners can be specifically defined on any graph coming from a metric space, where a heavy or undesirable network is given and finding a sparse and light-weight spanner and working with it instead of the actual network makes the computation easier and faster (Figure 1). In particular, lightweight spanners have been gained extreme attention in the geometric setting and in the metrics with bounded doubling dimension [29, 7, 6], which is a generalization of the former. The problem of finding sparse light-weight spanners in these spaces has appeared in many areas of computer science, including communication network design and distributed computing. These subgraphs have few edges and are easy to construct, leading them to appear in a wide range of applications since they were introduced [11, 34, 43]. In wireless ad hoc networks t-spanners are used to design sparse networks with guaranteed connectivity and guaranteed bounds on routing length [3]. In distributed computing spanners provide communication-efficiency and time-efficiency through the sparsity and the bounded stretch property [5, 19, 4, 20]. There has also been extensive use of geometric spanners in the analysis of road networks [21, 1, 10]. In robotics, geometric spanners helped motion planners to design near-optimal plans on a sparse and light subgraph of the actual network [16, 40, 15]. Spanners have many other applications including computing almost shortest paths [17, 12, 44, 26], and overlay networks [8, 47, 32].



(a) Complete graph. **(b)** 2-spanner. **(c)** 1.2-spanner. **(d)** 1.05-spanner.

Figure 1 A comparison of the complete graph on 30 random points on the plane with spanners of stretch 2, 1.2, and 1.05 on the same point set.

The special case where the underlying graph is a unit ball graph is motivated by the application of unit ball graphs in modeling wireless and ad-hoc networks, where the communication of the nodes are limited by their physical distances. The problem of finding sparse lightweight spanners for unit ball graphs in this settings translates into efficient topology control algorithms. Thus the necessity of a connected and energy-efficient topology for high-level routing protocols led researchers to develop many spanning algorithms for ad-hoc networks and in particular, UDGs. But the decentralized nature of ad-hoc networks demands that these algorithms be local instead of centralized. In these applications, it is important that the resulting topology is connected, has a low weight, and has a bounded degree, implying also that the number of edges is linear in the number of vertices.

Several known *proximity graphs* have been studied for this purpose, including the relative neighborhood graph (RNG), Gabriel graph (GG), Delaunay graph (DG), and Yao graph (YG). It is well-known that these proximity graphs are sparse and they can be calculated locally, using only the information from a node's neighborhood. But further analysis shows that they have poor bounds on at least one of the important criteria: maximum vertex degree, total weight, and stretch-factor [39].

Researchers have modified these constructions to fulfill the requirements. Li, Wan, and Wang [39] introduced a modified version of the Yao graph to resolve the issue of unbounded in-degree while preserving a small stretch-factor, but they left as an open question whether there exists a construction whose total weight is also bounded by a constant factor of the weight of the minimum spanning tree. The localized Delaunay triangulation (LDT) [38] and local minimum spanning tree (LMST) [37] were two other efforts in this way which failed to bound the total weight of the spanner. Hence bounding the weight became the main challenge in designing efficient spanners. The commonly used measure for the weight of the spanners is *lightness*, which is defined as the weight of the spanner divided by the weight of the minimum spanning tree.

In the distributed setting in particular, Gao, Guibas, Hershberger, Zhang, and Zhu [28] introduced restricted Delaunay graph (RDG), a planar distributed spanner construction for unit disk graphs in the two dimensional Euclidean plane that possessed a constant stretch-factor, leaving the weight of the spanner unstudied. Later Kanj, Perković, and Xia [33] presented the first local spanner construction for unit disk graphs in the two dimensional Euclidean plane, which also was planar and had constant bounds on its stretch-factor, maximum degree, and lightness. Their construction was also based on the Delaunay triangulation of the point set and required information from k-th hop neighbors of every node, for some constant k that depended on the input parameters.

In 2006, Damian, Pandit, and Pemmaraju [14] designed a distributed construction for $(1+\epsilon)$-spanners of the UBGs lying in *d-dimensional Euclidean space*. Their algorithm ran in $\mathcal{O}(\log^* n)$ rounds of communication and produced a $(1+\epsilon)$-spanner with constant bounds on its maximum degree and lightness. They used the so-called *leapfrog property* to prove the constant bound on the lightness of the spanner, which does not hold for the spaces of bounded doubling dimension in general. Instead, they showed in another work [13] that the weight of their spanner in the spaces of bounded doubling dimension is bounded by a factor $\mathcal{O}(\log \Delta)$ of the weight of the minimum spanning tree, where Δ is the ratio of the length of the longest edge in the unit ball graph divided by the length of its shortest edge. Besides these, their algorithm requires the knowledge of $\mathcal{O}(\frac{1}{\alpha-1})$-hop neighborhood of the nodes, which is costly in the CONGEST model of distributed computing, the more accepted and practical model than the LOCAL model of computation.

In the 3D Euclidean space, Jenkins, Kanj, Xia, and Zhang [31] designed the first localized bounded-degree $(1+\epsilon)$-spanner for unit ball graphs. They also presented a lightweight construction which possessed constant bounds on its stretch-factor and maximum degree. These algorithms again required k-th hop neighborhood information for every node, for a constant k that depended on the input parameters. Although these constructions were local, i.e. they ran in constant rounds of communication, they relied heavily on Euclidean transformations which made them inapplicable for other metric spaces.

Finally, Elkin, Filtser, and Neiman [18] studied the topic of lightweight spanners for general graphs and doubling graphs in the CONGEST model of distribution. For general graphs, they presented $(2k-1)\cdot(1+\epsilon)$-spanners with lightness $\mathcal{O}(k \cdot n^{1/k})$ in $\tilde{\mathcal{O}}(n^{0.5+1/(4k+2)}+D)$ rounds, where n is the number of vertices and D is the hop-diameter of the graph. For doubling graphs, they presented a $(1+\epsilon)$-spanner with lightness $\epsilon^{-\mathcal{O}(1)} \log n$ in $(\sqrt{n}+D)\cdot n^{o(1)}$ rounds of communication. Although these constructions are more general than the constructions of [13] and they perform in a more restricted model (CONGEST), they do not imply a superior result in the specific case of unit ball graphs in doubling metrics.

Apart from being a generalization of the Euclidean space, the importance of the spaces of bounded doubling dimension comes from the fact that a small perturbation in the pairwise distances does not affect the doubling dimension of the point set by much, while it can

change their Euclidean dimension significantly, or the resulting distances might not even be embeddable in Euclidean metrics at all [9]. This makes these metrics of bounded doubling dimension to be more applicable in real-world scenarios. On the other hand, geometric arguments are considered as a strong tool for proofs of sparsity and lightness bounds in Euclidean spaces, but in doubling spaces the only available tool besides metric properties, is the packing argument which is directly followed from the definition of the doubling dimension. Therefore, the sparsity and lightness results are more difficult to achieve in the spaces of bounded doubling dimension.

Since the work of Damian, Pandit, and Pemmaraju [13] in 2006, it has remained open whether UBGs in the spaces of bounded doubling dimension possess lightweight bounded-degree $(1+\epsilon)$-spanners and whether they can be found efficiently in a distributed model of computation. On the other hand, the construction of [13] requires complete information about the nodes in $\mathcal{O}(\frac{1}{\alpha-1})$ hops away, for some constant α. Acquiring this information is costly in the CONGEST model of computation, which is a more accepted model in distributed computing. Therefore, another open question arising from this line of work is to study the round complexity of the aforementioned problem in the CONGEST model. In this paper, we resolve both of these long-standing open questions by presenting centralized and distributed algorithms, both in the LOCAL, and the CONGEST model, for the purpose of finding such spanners.

1.1 Contributions

We have two main contributions in this paper. First, we resolve the proposed open question that has remained open for more than a decade, and we prove the existence of light-weight bounded-degree $(1+\epsilon)$-spanners of unit ball graphs in the spaces of bounded doubling dimension. Our construction has constant bounds on its maximum degree and its lightness, and it can be built in $\mathcal{O}(\log^* n)$ rounds of communication in the LOCAL model of computation, where n is the number of vertices.

Second, we propose the first lightweight spanner construction for unit ball graphs in the CONGEST model of computation. Even if we restrict our scope to the two dimensional Euclidean plane, where we see most of the applications of unit disk graphs, prior to this work there was known CONGEST algorithm for finding light spanners of unit disk graphs. We achieve this construction by making adjustments on our construction for the LOCAL model to make it work in the CONGEST model in the same asymptotic number of rounds. The bounds on the lightness and maximum degree of our spanner remain the same in this model.

Besides these main results, we modify these constructions for the two dimensional Euclidean plane in order to have a linear number of edge intersections in total, implying a constant average number of edge intersections per node. This is motivated by the observation that a higher intersection per edge causes a higher chance of interference between the corresponding endpoints. To the best of our knowledge, this is the first distributed low-stretch low-intersection spanner construction for unit disk graphs.

A more detailed version of our results can be found in the following theorems. First, we introduce a centralized algorithm Centralized-Spanner that,

▶ **Theorem 8.** *Given a weighted unit ball graph G in a metric of bounded doubling dimension and a constant $\epsilon > 0$, the spanner returned by Centralized-Spanner(G,ϵ) is a $(1+\epsilon)$-spanner of G and has constant bounds on its lightness and maximum degree. These constant bounds only depend on ϵ and the doubling dimension.*

We use this centralized construction to propose the distributed construction Distributed-Spanner in the LOCAL model of computation,

▶ **Theorem 16.** *Given a weighted unit ball graph G with n vertices in a metric of bounded doubling dimension and a constant $\epsilon > 0$, the algorithm Distributed-Spanner(G,ϵ) runs in $\mathcal{O}(\log^* n)$ rounds of communication in the LOCAL model of computation, and returns a $(1+\epsilon)$-spanner of G that has constant bounds on its lightness and maximum degree. These constant bounds only depend on ϵ and the doubling dimension.*

Next, we study the problem in the CONGEST model of computation. Our distributed construction Distributed-Spanner requires complete information about 2-hop neighborhood of a selected set of vertices, which is not easy to acquire in the CONGEST model. The same issues exists in the distributed algorithm of [13], where they aggregate information about the nodes that are $\mathcal{O}(\frac{1}{\alpha-1})$ hops away, for some constant α. A simple approach for aggregating 2-hop neighborhoods would require $\mathcal{O}(d)$ rounds of communication in the CONGEST model, which can be as large as $\Omega(n)$ if the input graph is dense. In our next theorem, we break this barrier by making some adjustments for our algorithm to work in the CONGEST model of computation. Despite adding to the complexity of the algorithm itself, we prove that the round complexity of our new algorithm, CONGEST-Spanner, would still be bounded by $\mathcal{O}(\log^* n)$.

▶ **Theorem 21.** *Given a weighted unit ball graph G with n vertices in a metric of bounded doubling dimension and a constant $\epsilon > 0$, the algorithm CONGEST-Spanner(G,ϵ) runs in $\mathcal{O}(\log^* n)$ rounds of communication in the CONGEST model of computation, and returns a $(1+\epsilon)$-spanner of G that has constant bounds on its lightness and maximum degree. These constant bounds only depend on ϵ and the doubling dimension.*

The rest of the contributions are included in the full version of this paper due to page limits. In the full version, We study the problem in the case of the two dimensional Euclidean plane, where the greedy spanner on a complete weighted graph is known to have constant upper bounds on its lightness [27], maximum degree, and average number of edge intersections per node [23]. We observe that a simple change on the this algorithm can extend these results for unit disk graphs as well. We call this modified algorithm Centralized-Euclidean-Spanner and we show that

▶ **Theorem 22.** *Given a weighted unit disk graph G in the two dimensional Euclidean plane and a constant $\epsilon > 0$, the spanner returned by Centralized-Euclidean-Spanner(G,ϵ) is a $(1+\epsilon)$-spanner of G and has constant bounds on its lightness, maximum degree, and the average number of edge intersections per node. These constant bounds only depend on ϵ and the doubling dimension.*

We use the aforementioned construction to propose Distributed-Euclidean-Spanner, a specific distributed low-intersection construction for the case of the two dimensional Euclidean plane that preserves the previously mentioned properties and adds the low-intersection property.

▶ **Theorem 26.** *Given a weighted unit disk graph G with n vertices in the two dimensional Euclidean plane and a constant $\epsilon > 0$, the algorithm Distributed-Euclidean(G,ϵ) runs in $\mathcal{O}(\log^* n)$ rounds of communication and returns a bounded-degree $(1+\epsilon)$-spanner of G that has constant bounds on its lightness, maximum degree, and the average number of edge intersections per node. These constant bounds only depend on ϵ and the doubling dimension.*

We also prove that the last construction possesses sublinear separators and a separator hierarchy in the two dimensional Euclidean plane. We generalize this result to work for higher dimensions of Euclidean spaces. Finally, we provide experimental results on random point sets in the two dimensional Euclidean plane that confirm the efficiency of our distributed construction.

2 Preliminaries

2.1 Doubling metrics

We start by recalling the definition of the doubling dimension of a metric space,

▶ **Definition 1** (doubling dimension). *The doubling dimension of a metric space is the smallest d such that for any $R > 0$, any ball of radius R can be covered by at most 2^d balls of radius $R/2$.*

We say a metric space has *bounded doubling dimension* if its doubling dimension is upper bounded by a constant. Besides the triangle inequality, which is intrinsic to metric spaces, the *packing lemma* is an essential tool for the metrics of bounded doubling dimension. This lemma states that it is impossible to pack more than a certain number of points in a ball of radius $R > 0$ without making a pair of points' distance less than some $r > 0$.

▶ **Lemma 2** (Packing Property). *In a metric space of bounded doubling dimension d, let X be a set of points with minimum distance r, contained in a ball of radius R. Then $|X| \le \left(\frac{4R}{r}\right)^d$.*

Proof. This is a well-known fact, see e.g. [46]. ◀

2.2 Spanners for complete graphs

For a weighted graph G in a metric space, where every edge weight is equal to the metric distance of its endpoints, a t-spanner is defined in the following way,

▶ **Definition 3** (t-spanner). *A t-spanner of a weighted graph G is a subgraph S of G that for every pair of vertices x, y in G,*

$$\mathrm{dist}_S(x, y) \le t \cdot \mathrm{dist}_G(x, y)$$

where $\mathrm{dist}_A(x, y)$ is the length of a shortest path between x and y in A. The lightness of S is defined as $\mathbf{w}(S)/\mathbf{w}(MST)$ where $\mathbf{w}$ is the weight function and MST is the minimum spanning tree in G.

In other words, a t-spanner approximates the pairwise distances within a factor of t. Spanners were studied for complete weighted graphs first, and several constructions were proposed to optimize them with respect to the number of edges and total weight. Among these constructions, *greedy spanners* [2] are known to out-perform the others.

A greedy spanner (Figure 1) can be constructed by running the greedy spanner algorithm (Algorithm 1) on a set of points V in a metric space. This short procedure adds edges one at a time to the spanner it constructs, in ascending order by length. For each pair of vertices, in this order, it checks whether the pair already satisfies the distance inequality using the edges already added. If not, it adds a new edge connecting the pair. Therefore, by construction, each pair of vertices satisfies the inequality, either through previous edges or (if not) through the newly added edge. The resulting graph is therefore a t-spanner.

Algorithm 1 The naive greedy spanner algorithm.

```
1: procedure NAIVE-GREEDY(V)
2:     Let S be a graph with vertices V and edges E = {}
3:     for each pair (P, Q) ∈ V^2 in increasing order of ‖PQ‖ do
4:         if dist_S(P, Q) > t · dist(P, Q) then
5:             Add edge PQ to E
       return S
```

Despite the simplicity of Algorithm 1, Farshi and Gudmundsson [25] observed that in practice, greedy spanners are surprisingly good in terms of the number of edges, weight, maximum vertex degree, and also the number of edge crossings in the two dimensional Euclidean plane. All of these properties have been proven rigorously so far. Filster and Solomon [27] proved that greedy spanners have size and lightness that is optimal to within a constant factor for worst-case instances. They also achieved a near-optimality result for greedy spanners in spaces of bounded doubling dimension. Borradaile, Le, and Wulff-Nilsen [7] recently proved optimality for doubling metrics, generalizing a result of Narasimhan and Smid [41], and resolving an open question posed by Gottlieb [29], and Le and Solomon showed that no geometric t-spanner can do asymptotically better than the greedy spanner in terms of number of edges and lightness [36].

In a recent work, Eppstein and Khodabandeh [23] showed that the number of edge crossings of the greedy spanner in the two dimensional Euclidean plane is linear in the number of vertices. Moreover, they proved that the crossing graph of the greedy spanner has bounded degeneracy, implying the existence of sub-linear separators for these graphs [22]. This, together with the well-known fact that greedy spanners have bounded-degree in the two dimensional Euclidean plane, makes greedy spanners more practical in this particular metric space.

Although the degree of the greedy spanner is bounded in the two dimensional Euclidean plane, it is known that there exist n-point metric spaces with doubling dimension 1 where the greedy spanner has maximum degree $n - 1$ [27]. Gudmundsson, Levcopoulos, and Narasimhan [30] devised a faster algorithm that was later proven to have bounded degree as well as constant lightness and linear number of edges [27]. We call this algorithm APPROXIMATE-GREEDY in this paper, and we make use of it in our algorithms for the metrics of bounded doubling dimension, while we take advantage of the extra low-intersection property of NAIVE-GREEDY in the two dimensional Euclidean plane.

2.3 Unit ball graphs

We formally define a unit ball graph on a set of points V in the following way,

▶ **Definition 4** (unit ball graph). *Given a set of points V in a metric space, the unit ball graph G on V contains V as its vertex set and every two vertices $x, y \in V$ are connected if and only if $\|xy\| \leq 1$. The weight of an edge (x, y) is equal to $\|xy\|$ if the edge exists.*

Unit ball graphs are an important subclass of the graphs called *growth-bounded graphs*, which only limit the number of independent nodes in every neighborhood, a property that holds for UBGs due to the packing property.

Kuhn, Moscibroda, and Wattenhofer [35] presented a $\mathcal{O}(\log^* n)$-round distributed algorithm for finding a maximal independent set (MIS) of a unit ball graph graph in a space with bounded doubling dimension. This result was later generalized by Schneider and Wattenhofer [45] for growth-bounded graphs. Throughout the paper we refer to their algorithm by

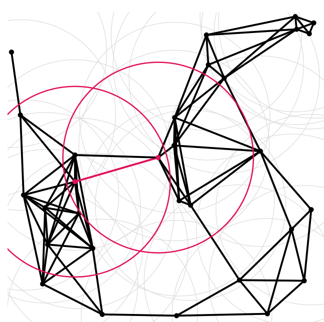


Figure 2 The unit disk graph on the same point set introduced earlier. The red disks intersect, therefore there is an edge between their centers.

Maximal-Independent. It turns out that Maximal-Independent will be a key ingredient of our distributed algorithms, as well as their bottleneck in terms of the number of rounds. This means that if a maximal independent set is known beforehand, our algorithms can be executed fully locally, in constant number of rounds.

In section 3 we prove the existence of $(1+\epsilon)$-spanners with constant bounds on the maximum degree and the lightness by introducing an algorithm that finds such spanners in a centralized manner. In section 4 we propose a distributed construction that delivers the same features through a $\mathcal{O}(\log^* n)$-round algorithm. In the full version, we consider the special case of two dimensional Euclidean plane and we design centralized and distributed algorithms to construct a spanner that has the extra low-intersection property, making it more suitable for practical purposes.

3 Centralized Construction

In this section we propose a centralized construction for a light-weight bounded-degree $(1+\epsilon)$-spanner for unit ball graphs in a metric of bounded doubling dimension. Later in section 4 we use this centralized construction to design a distributed algorithm that delivers the same features.

It is worth mentioning that the greedy spanner would be a $(1+\epsilon)$-spanner of the UBG if the algorithm stops after visiting the pairs of distance at most 1, and it even has a lightness bounded by a constant, but as we mentioned earlier, there are metrics with doubling dimension 1 in which its degree may be unbounded.

To construct a lightweight bounded-degree $(1+\epsilon)$-spanner of the unit ball graph, we start with the spanner of [30], called Approximate-Greedy, which is returns a spanner of the complete graph. It is proven in [41] that Approximate-Greedy has the desired properties, i.e. bounded-degree and lightness, for complete weighted graphs in Euclidean metrics, but as stated in [27], the proof only relies on the triangle inequality and packing argument which both work for doubling metrics as well. Therefore, we may safely assume that Approximate-Greedy finds a light-weight bounded-degree $(1+\epsilon)$-spanner of the complete weighted graph defined on the point set. The main issue is that the edges of length more than 1 are not allowed in a spanner of the unit ball graph on the same point set. Therefore, a replacement procedure is needed to substitute these edge with edges of length at most 1. Peleg and Roditty [42] introduced a refinement process which removes the edges of length larger than 1 from the spanner and replaces them with three smaller edges to make the output a subgraph of the UBG. The main issue with their approach is that it can lead to

vertices having unbounded degrees in the spanner, therefore missing an important feature. Here, we introduce our own refinement process that not only replaces edges of larger than 1 with smaller edges and makes the spanner a subgraph of the unit ball graph, but also guarantees a constant bounded on the degrees of the resulting spanner.

3.1 The algorithm

In the first step of the algorithm (Algorithm 2) we choose $\epsilon' = \epsilon/36$, a smaller stretch parameter than ϵ, to cover the errors that future steps might inflict to the spanner. Then we call the procedure Approximate-Greedy on the set of vertices V to calculate a light-weight bounded-degree $(1+\epsilon')$-spanner S of the complete weighted graph on V. This spanner might contain edges of length larger than 1, which we will replace by some edges of length at most 1 in the future steps.

Since an edge of length larger than $1+\epsilon'$ in S cannot participate in the shortest path between any two adjacent vertices in G, we simply remove and discard them from the spanner. Then for every remaining edge $e = (u, v)$ of length in the range $(1, 1+\epsilon']$ we find an edge (x, y) of the original graph G so that $\|ux\| \leq \epsilon'$ and $\|vy\| \leq \epsilon'$. We then replace such an edge e by the edge (x, y). We call the pair (x, y) the *replacement edge* or the *replacement pair* for the edge e. Since this procedure can end up assigning too many replacement edges to a single vertex (x or y in this case) and hence increasing its degree significantly, we perform a simple check before adding a replacement edge; we store the set R of previously added replacement pairs in the memory and if a *weak* replacement pair $(x', y') \in R$ exists, then we prefer it over a newly found replacement pair $(x, y) \notin R$. By weak replacement pair we mean a pair $(x', y') \in R$ that $\|ux'\| \leq 2\epsilon'$ and $\|vy'\| \leq 2\epsilon'$, which is weaker than the definition of the replacement pair. As we later see this weaker notion of replacement pair will help us to bound the degree of the vertices.

After removing the edges of length larger than 1 and replacing the ones in the range $(1, 1+\epsilon']$, we return the spanner to the output.

Algorithm 2 A centralized spanner construction.

Input. A unit ball graph $G(V, E)$ in a metric with doubling dimension d.
Output. A light-weight bounded-degree $(1+\epsilon)$-spanner of G.

```
1:  procedure Centralized-Spanner(G, ϵ)
2:      ϵ' ← ϵ/36
3:      S ← Approximate-Greedy(V, ϵ')
4:      R ← ∅
5:      for e = (u, v) in S do
6:          if |e| > 1 then
7:              Remove e from S
8:          if |e| ∈ (1, 1 + ϵ'] then
9:              if ∃(x, y) ∈ E that ‖ux‖ ≤ ϵ' and ‖vy‖ ≤ ϵ' then
10:                 if ∄(x', y') ∈ R that ‖ux'‖ ≤ 2ϵ' and ‖vy'‖ ≤ 2ϵ' then
11:                     S ← S ∪ {(x, y)}
12:                     R ← R ∪ {(x, y)}
13:     return S
```

3.2 The analysis

Now we prove that the output S of the algorithm is a light-weight bounded-degree $(1+\epsilon)$-spanner of the unit ball graph G. Clearly, after the refinement is done the spanner S is a subgraph of G, so we need to analyze the lightness, the stretch factor, and the maximum degree of the spanner.

First we prove that the stretch-factor of the spanner is indeed bounded by $1+\epsilon$.

▶ **Lemma 5.** *The spanner returned by* Centralized-Spanner *has a stretch factor of* $1+\epsilon$*.*

The proof of this lemma is moved to Appendix A. Now we analyze the weight of the spanner, proving its constant lightness.

▶ **Lemma 6.** *The spanner returned by* Centralized-Spanner *has a weight of* $\mathcal{O}(1)\mathbf{w}(MST)$*.*

The proof of this lemma is also in Appendix A. In the final step, we bound the maximum degree of the spanner.

▶ **Lemma 7.** *The spanner returned by* Centralized-Spanner *has bounded degree.*

The proof of this lemma is moved to Appendix A. Putting these together, we can prove Theorem 8.

▶ **Theorem 8** (Centralized Spanner)**.** *Given a weighted unit ball graph G in a metric of bounded doubling dimension and a constant $\epsilon > 0$, the spanner returned by* Centralized-Spanner*(G,ϵ) is a $(1+\epsilon)$-spanner of G and has constant bounds on its lightness and maximum degree. These constant bounds only depend on ϵ and the doubling dimension.*

Proof. Follows directly from Lemma 5, Lemma 6, and Lemma 7. ◀

4 Distributed Construction

In this section we propose our distributed construction for finding a $(1+\epsilon)$-spanner of a unit ball graph using only 2-hop neighborhood information. The spanner returned by our algorithm has constant bounds on its maximum degree and its lightness. This is the first light-weight distributed construction for unit ball graphs in doubling metrics, to the best of our knowledge.

In our distributed construction, we run our centralized algorithm on the 2-hop neighborhoods of a an independent set of the unit ball graph, and we prove that putting these local spanners together will achieve a spanner that possesses the desired properties.

4.1 The algorithm

For the distributed construction we propose Algorithm 3. There is a preprocessing step of finding a maximal independent set I of G, which can be done using the distributed algorithm of [35] in $\mathcal{O}(\log^* n)$ rounds. We refer to this algorithm by Maximal-Independent. Then the Local-Greedy subroutine is run on every vertex $w \in I$ to find a $(1+\epsilon)$-spanner S_w of the 2-hop neighborhood of w, denoted by $\mathcal{N}^2(w)$. At the final step, every $w \in I$ sends its local spanner edges to the corresponding endpoints of every edge. Symmetrically, every vertex listens for the edges sent by the vertices in I and once a message is received, it stores the edges in its local storage. In other words, the final spanner is the union of all these local spanners. We use the centralized algorithm of section 3 for every local neighborhood $\mathcal{N}^2(w)$ to guarantee the bounds that we need.

Algorithm 3 The localized greedy algorithm.

Input. A unit ball graph $G(V, E)$ in a metric with doubling dimension d and an $\epsilon > 0$.
Output. A light-weight bounded-degree $(1 + \epsilon)$-spanner of G.

```
1: procedure DISTRIBUTED-SPANNER(G, ε)
2:     Find a maximal independent set I of G using [35]
3:     Run LOCAL-GREEDY on the vertices of G
4: function LOCAL-GREEDY(vertex w)
5:     Retrieve N^2(w), the 2-hop neighborhood information of w
6:     if w ∈ I then
7:         S_w ← CENTRALIZED-SPANNER(N^2(w), ε)
8:         for e = (u, v) in S_w do
9:             Send e to u and v
10:    Listen to incoming edges and store them
```

Similar to the aforementioned greedy algorithm (Algorithm 1), our algorithm seems very simple in the first sight. But as we see later in this section, proving its properties, particularly its lightness, is a non-trivial task.

4.2 The analysis

Now we show that the spanner introduced in Algorithm 3 possesses the desired properties. First, we show the round complexity of $\mathcal{O}(\log^* n)$.

▶ **Lemma 9.** *Distributed-Spanner can be done in $\mathcal{O}(\log^* n)$ rounds of communication.*

The proof of this lemma is in Appendix B. Next we bound the stretch-factor of the spanner.

▶ **Lemma 10.** *The spanner returned by Distributed-Spanner has a stretch factor of $1 + \epsilon$.*

The proof of this lemma is also included in Appendix B. Now we bound the maximum degree of the spanner.

▶ **Lemma 11.** *The spanner returned by Distributed-Spanner has a bounded degree.*

The proof of this lemma is also in Appendix B. In order to bound the lightness of the output, we assume that $\epsilon \leq 1$ and we make a few comparisons. First, for any $w \in I$ we compare the weight of $\mathcal{S}_w$ to the weight of the minimum spanning tree on $\mathcal{N}^2(w)$. Then we compare the weight of the minimum spanning tree on $\mathcal{N}^2(w)$ to the weight of the minimum Steiner tree on $\mathcal{N}^3(w)$, where the required vertices are $\mathcal{N}^2(w)$ and 3-hop vertices are optional. Finally, we compare the weight of this minimum Steiner tree to the weight of the induced subgraph of Centralized-Spanner(G, ϵ) on the subset of vertices $\mathcal{N}^3(w)$, which later implies that the overall weight of $\mathcal{S}_w$s is bounded by a constant factor of the weight of the minimum spanning tree on G.

Our first claim is that the weight of $\mathcal{S}_w$ is bounded by a constant factor of the weight of the MST on $\mathcal{N}^2(w)$.

▶ **Corollary 12.** $\mathbf{w}(\mathcal{S}_w) = \mathcal{O}(1)\mathbf{w}(MST(\mathcal{N}^2(w)))$

Proof. Follows from the properties of the centralized algorithm in section 3. ◀

Next we compare $\mathbf{w}(MST(\mathcal{N}^2(w)))$ to the weight of the minimum Steiner tree of $\mathcal{N}^3(w)$ on the required vertices $\mathcal{N}^2(w)$.

▶ **Lemma 13.** *Define $\mathcal{T}$ to be the optimal Steiner tree on the set of vertices $\mathcal{N}^3(w)$, where only vertices in $\mathcal{N}^2(w)$ are required and the rest of them are optional. Then*

$$\mathbf{w}(MST(\mathcal{N}^2(w))) \leq 2\mathbf{w}(\mathcal{T})$$

The proof of this lemma is included in Appendix B. We then compare the weight of $\mathcal{T}$ to the weight of induced subgraph of CENTRALIZED-SPANNER(G, ϵ) on the subset of vertices $\mathcal{N}^3(w)$. The main observation here is that when $\epsilon \leq 1$ the induced subgraph of the centralized spanner on $\mathcal{N}^3(w)$ would be a feasible solution to the minimum Steiner tree problem on $\mathcal{N}^3(w)$, with the required vertices being the vertices in $\mathcal{N}^2(w)$. This will imply that the weight of the induced subgraph is at least equal to the weight of the minimum Steiner tree.

▶ **Lemma 14.** *Let $\mathcal{S}^*$ be the output of CENTRALIZED-SPANNER(G, ϵ) and let $\mathcal{S}^*_w$ be the induced subgraph of $\mathcal{S}^*$ on $\mathcal{N}^3(w)$. Then*

$$\mathbf{w}(\mathcal{T}) \leq \mathbf{w}(\mathcal{S}^*_w)$$

The proof of this lemma is also in Appendix B. This lemma concludes our sequence of comparisons. By putting together what we proved so far, we have

▶ **Proposition 15.** *The spanner returned by DISTRIBUTED-SPANNER has a weight of $\mathcal{O}(1)\mathbf{w}(MST)$.*

Proof. By Corollary 12, Lemma 13, and Lemma 14,

$$\mathbf{w}(\mathcal{S}_w) = \mathcal{O}(1)\mathbf{w}(\mathcal{S}^*_w)$$

Summing up together these inequalities for $w \in I$,

$$\mathbf{w}(\text{output}) = \mathcal{O}(1)\sum_{w\in I}\mathbf{w}(\mathcal{S}^*_w)$$

But we recall that every vertex, and hence every edge of $\mathcal{S}^*$, is repeated $\mathcal{O}(1)$ times in the summation above, so

$$\mathbf{w}(\text{output}) = \mathcal{O}(1)\mathbf{w}(\mathcal{S}^*) = \mathcal{O}(1)\mathbf{w}(MST(G))$$ ◀

Therefore we have all the ingredients to prove Theorem 16.

▶ **Theorem 16** (Distributed Spanner)**.** *Given a weighted unit ball graph G with n vertices in a metric of bounded doubling dimension and a constant $\epsilon > 0$, the algorithm DISTRIBUTED-SPANNER(G,ϵ) runs in $\mathcal{O}(\log^* n)$ rounds of communication in the LOCAL model of computation, and returns a $(1+\epsilon)$-spanner of G that has constant bounds on its lightness and maximum degree. These constant bounds only depend on ϵ and the doubling dimension.*

Proof. It directly follows from Lemma 9, Lemma 10, Lemma 11, and Proposition 15. ◀

5 Adjustments for the CONGEST Model

In this section we study the problem of finding a bounded-degree $(1+\epsilon)$-spanner in the CONGEST model of computation, for a point set that is located in a doubling metric space. In the CONGEST model, every node can send a message of bounded size to every other node in a single round of communication. This makes it hard to gather any global information about the graph.

The maximal independent set algorithm of [35] still works in $\mathcal{O}(\log^* n)$ rounds of communication in the CONGEST model. But our proposed distributed algorithm (Algorithm 3) needs to gather 2-hop neighborhood information of every center in the MIS, which requires $\mathcal{O}(D)$ rounds in the CONGEST model, where D is the maximum degree of a vertex in the unit ball graph. The rest of the algorithm is performed locally and the number edges sent to every neighbor in the end is bounded by a constant, so the remaining of the algorithm only requires a constant number of rounds.

It is natural to ask whether our algorithm can be adapted to the CONGEST model, and if it requires more communication rounds compared to the LOCAL model. In this section we show how to modify our algorithm to work in the CONGEST model, and surprisingly, have no asymptotic change on its number of communication rounds.

As we mentioned earlier, the only step of our algorithm that requires more than constant rounds of communication is the aggregation of the 2-hop neighborhood information for every center in the MIS. We passed the 2-hop neighborhoods to our centralized algorithm to find an asymptotically optimal spanner on them, which was later distributed among the vertices in the neighborhood to form the final spanner. Here, for our construction in the CONGEST model, we directly address the problem of finding an asymptotically optimal spanner for the 2-hop neighborhoods, without the need to access all of the points in those neighborhoods.

Let $w \in I$ be a center in the maximal independent set. We partition the edges of the UBG on $\mathcal{N}^2(w)$ into two sets, depending on whether their length is larger than $1/2$ or not. We aim to find asymptotically optimal spanners for each partition separately. We use the notation $G_{\leq\alpha}$ to refer to the subgraph of the unit ball graph that consists of edges of length at most α. We similarly define $G_{>\alpha}$. Therefore, we can refer to the subgraphs induced by the two partitions by $G_{\leq 1/2}$ and $G_{>1/2}$.

First, we show that in constant rounds of communication, we can find a covering of the points in $\mathcal{N}^2(w)$ with at most a constant number of balls of radius $1/2$. The existence of such covering trivially follows from the definition of a doubling metric space, but finding such covering in the distributed setting is not trivial. Therefore, we introduce the following procedure: Every center $v \in \mathcal{N}^1(w)$ (including w itself) finds a maximal independent set $I_{1/4}(v)$ of the vertices $\mathcal{N}^1(v)$ in $G_{\leq 1/4}$, and sends it to w, all centers at the same time. Recall that $\mathcal{N}^1(v)$ is the set of neighbors of v in the UBG, and a maximal independent set in $G_{\leq 1/4}$ is simply a maximal set of vertices where the pair-wise distance of each two vertex is at least $1/4$. Finding this maximal independent set for each v can be easily done using a (centralized) greedy algorithm, and the size of such maximal independent set would be bounded by a constant according to the packing lemma. Therefore, this step can be done in constant number of rounds. Afterwards, w calculates a maximal independent set $\mathcal{I}(w)$ of the vertices $\cup_{v\in\mathcal{N}^1(w)} I_{1/4}(v)$ in $G_{\leq 1/4}$. We show that the centers in $\mathcal{I}$ satisfy our desired properties.

▶ **Lemma 17.** *The union of the balls of radius $1/2$ around the centers in $\mathcal{I}(w)$ cover $\mathcal{N}^2(w)$. Furthermore, the size of $\mathcal{I}(w)$ is bounded by a constant.*

The proof of this lemma is moved to Appendix C. Next, every center $v \in \mathcal{I}(w)$ calculates a $(1+\epsilon)$-spanner $\mathcal{S}_{\leq 1/2}(v)$ of the point set $\mathcal{N}^1(v)$ using the centralized algorithm, and notifies its neighbors about their connections. We prove that the union of these spanners, would be a spanner for one of the partitions, i.e. the edges of length at most $1/2$ in $\mathcal{N}^2(w)$. The pseudo-code of this procedure is available in Algorithm 5

▶ **Lemma 18.** *The union of the spanners $\mathcal{S}_{\leq 1/2}(v)$ for $v \in \mathcal{I}(w)$ is a $(1+\epsilon)$-spanner of $\mathcal{N}^2(w)$ in $G_{\leq 1/2}$. The maximum degree and the lightness of this spanner are both bounded by constants.*

The proof of this lemma can be found in Appendix C.

Algorithm 4 The CONGEST spanner algorithm.

Input. A unit ball graph $G(V, E)$ in a metric with doubling dimension d and an $\epsilon > 0$.
Output. A light-weight bounded-degree $(1+\epsilon)$-spanner of G.

```
1: procedure CONGEST-SPANNER(G, ε)
2:     Find a maximal independent set I of G using [35]
3:     Run SPAN-SHORT-EDGES on the vertices of G
4:     Run SPAN-LONG-EDGES on the vertices of G
```

Algorithm 5 Finding a spanner of the edges of length smaller than $1/2$ in $\mathcal{N}^2(w)$.

```
1: function SPAN-SHORT-EDGES(vertex u)
2:     if u ∈ I then
3:         Send a signal of type 1 to every v ∈ N^1(u).
4:         Wait for their maximal independent sets, I_{1/4}(v)s.
5:         Calculate a maximal independent set of ∪_{v∈N^1(u)} I_{1/4}(v) in G_{<1/4} greedily.
6:         Store this maximal independent set in I(u).
7:         Send a signal of type 2 to every v ∈ I(u).
8:     if received type 1 signal from some w then
9:         Calculate a maximal independent set of N^1(w) in G_{1/4} greedily.
10:        Send this maximal independent set to w.
11:    if received type 2 signal from some w then
12:        Calculate S_{≤1/2}(u) ← CENTRALIZED-SPANNER(N^1(u), ε)
13:        for e = (a, b) in S_{≤1/2}(u) do
14:            Send e to a and b
15:    Receive and store the edges sent by other centers
```

Now we find a spanner for the other partition, the edges of length larger than $1/2$ in $\mathcal{N}^2(w)$. The procedure is as follows: First, every center $v \in \mathcal{N}^1(w)$ calculates a maximal independent set $I_{\epsilon/40}(v)$ of $\mathcal{N}^1(v)$ in $G_{\leq \epsilon/40}$ and sends it to w. Again, the size of each maximal independent set is $\mathcal{O}(\epsilon^{-d})$ by the packing lemma, which is constant. Therefore, this step takes only constant number of rounds. Afterwards, w finds a maximal independent set $\mathcal{I}'(w)$ of $\cup_{v \in \mathcal{N}^1(w)} I_{\epsilon/40}(v)$ in $G_{\leq \epsilon/40}$. Then w constructs a $(1+\epsilon/5)$-spanner $\mathcal{S}'(w)$ of $\mathcal{I}'(w)$ in G using the centralized algorithm, and announces the edges of the spanner to their corresponding endpoints. Finally, every center $v \in \mathcal{I}'(w)$ calculates a $(1+\epsilon)$-spanner $\mathcal{S}''(v)$ of its $\epsilon/20$ neighborhood and announces its edges to their endpoints. We show that the union of $S'(w)$ and $S''(v)$s for $v \in \mathcal{I}'(w)$ would form a $(1+\epsilon)$-spanner of the second partition, i.e. the edges of larger than $1/2$. The pseudo-code of this procedure is available in Algorithm 6

▶ **Lemma 19.** *The union of the balls of radius $\epsilon/20$ around the centers in $\mathcal{I}'(w)$ cover $\mathcal{N}^2(w)$. Furthermore, the size of $\mathcal{I}'(w)$ is bounded by a constant.*

Proof. Similar to the proof of Lemma 17. ◀

▶ **Lemma 20.** *The union of the spanners $\mathcal{S}'(w)$ and $\mathcal{S}''(v)$ for $v \in \mathcal{I}'(w)$ forms a $(1+\epsilon)$-spanner of $\mathcal{N}^2(w)$ in $G_{>1/2}$. The maximum degree and the lightness of this spanner are btoh bounded by constants.*

The proof of this lemma is also in Appendix C.

Algorithm 6 Finding a spanner of the edges of length larger than $1/2$ in $\mathcal{N}^2(w)$.

1: **function** Span-Long-Edges(vertex u)
2: **if** $u \in I$ **then**
3: Send a signal of type 3 to every $v \in \mathcal{N}^1(u)$.
4: Wait for their maximal independent sets, $I_{\epsilon/40}(v)$s.
5: Calculate a maximal independent set of $\cup_{v\in\mathcal{N}^1(u)} I_{\epsilon/40}(v)$ in $G_{<\epsilon/40}$ greedily.
6: Store this maximal independent set in $\mathcal{I}'(u)$.
7: Send a signal of type 4 to every $v \in \mathcal{I}'(u)$.
8: Calculate $\mathcal{S}'(u) \leftarrow$ Centralized-Spanner($\mathcal{I}'(u)$, $\epsilon/5$)
9: **for** $e = (a, b)$ in $\mathcal{S}'(u)$ **do**
10: Send e to a and b
11: **if** received type 3 signal from some w **then**
12: Calculate a maximal independent set of $\mathcal{N}^1(w)$ in $G_{\epsilon/40}$ greedily.
13: Send this maximal independent set to w.
14: **if** received type 4 signal from some w **then**
15: Let $\mathcal{N}^{\epsilon/20}(u)$ be the $\epsilon/20$ neighborhood of u, i.e. the set of vertices that are at distance $\epsilon/20$ or less from u.
16: Calculate $\mathcal{S}''(u) \leftarrow$ Centralized-Spanner($\mathcal{N}^{\epsilon/20}(u)$, ϵ)
17: **for** $e = (a, b)$ in $\mathcal{S}''(u)$ **do**
18: Send e to a and b
19: Receive and store the edges sent by other centers

The union of the two spanners for the two partitions form a spanner for the 2-hop neighborhood of w, the goal we wanted to achieve in the CONGEST model. This completes our adjustments in this model.

▶ **Theorem 21** (CONGEST Spanner). *Given a weighted unit ball graph G with n vertices in a metric of bounded doubling dimension and a constant $\epsilon > 0$, the algorithm CONGEST-Spanner(G,ϵ) runs in $\mathcal{O}(\log^* n)$ rounds of communication in the CONGEST model of computation, and returns a $(1+\epsilon)$-spanner of G that has constant bounds on its lightness and maximum degree. These constant bounds only depend on ϵ and the doubling dimension.*

Proof. The proof follows from Lemma 18 and Lemma 20. ◀

We defer our low-intersection construction as well as our experimental results to the full version of the paper due to space limitation.

6 Conclusions

In this paper we resolved an open question from 2006 and we proved the existence of light-weight bounded-degree $(1+\epsilon)$-spanners for unit ball graphs in the spaces of bounded doubling dimension. Moreover, we provided a centralized construction and a distributed construction in the LOCAL model that finds a spanner with these properties. Our distributed construction runs in $\mathcal{O}(\log^* n)$ rounds, where n is the number of vertices in the graph. If a maximal independent set of the unit ball graph is known beforehand, our algorithm runs in constant number of rounds. Next, we showed how to adjust our distributed construction to work in the CONGEST model, without touching its asymptotic round complexity. In this way, we provided the first CONGEST algorithm for finding a light spanner of unit ball graphs.

In the full version, we further adjusted these algorithms for the case of unit disk graphs in the two dimensional Euclidean plane, and we presented the first centralized and distributed constructions for a light-weight bounded-degree $(1+\epsilon)$-spanner that also has a linear number of edge intersections in total. This can be useful for practical purposes if minimizing the number of edge intersections is a priority. We proved, based on this low-intersection property, that our spanner has sub-linear separators, and a separator hierarchy, and we were able to generalize this result to higher dimensions of Euclidean spaces.

Finally, we performed experiments (in the full version) on random point sets in the two dimensional Euclidean plane, to ensure that our theoretical bounds are also supported by enough empirical evidence. Our results show that our construction performs efficiently with respect to the maximum degree, size, and total weight.

References

1 Mohammad Ali Abam, Mark De Berg, Mohammad Farshi, and Joachim Gudmundsson. Region-fault tolerant geometric spanners. *Discrete & Computational Geometry*, 41(4):556–582, 2009. doi:10.1007/s00454-009-9137-7.

2 Ingo Althöfer, Gautam Das, David Dobkin, and Deborah Joseph. Generating sparse spanners for weighted graphs. In John R. Gilbert and Rolf Karlsson, editors, *Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 447 of *Lecture Notes in Computer Science*, pages 26–37. Springer, 1990. doi:10.1007/3-540-52846-6_75.

3 Khaled Alzoubi, Xiang-Yang Li, Yu Wang, Peng-Jun Wan, and Ophir Frieder. Geometric spanners for wireless ad hoc networks. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):408–421, 2003. doi:10.1109/TPDS.2003.1195412.

4 Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Near-linear time construction of sparse neighborhood covers. *SIAM Journal on Computing*, 28(1):263–277, 1998. doi:10.1137/S0097539794271898.

5 Surender Baswana, Telikepalli Kavitha, Kurt Mehlhorn, and Seth Pettie. Additive spanners and (α, β)-spanners. *ACM Transactions on Algorithms*, 7(1):5, 2010. doi:10.1145/1868237.1868242.

6 Sujoy Bhore, Arnold Filtser, Hadi Khodabandeh, and Csaba D Tóth. Online spanners in metric spaces. *arXiv preprint*, 2022. arXiv:2202.09991.

7 Glencora Borradaile, Hung Le, and Christian Wulff-Nilsen. Greedy spanners are optimal in doubling metrics. In *Proceedings of the 30th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2371–2379. Society for Industrial and Applied Mathematics, 2019. doi:10.1137/1.9781611975482.145.

8 Rebecca Braynard, Dejan Kostic, Adolfo Rodriguez, Jeffrey Chase, and Amin Vahdat. Opus: an overlay peer utility service. In *Proceedings of the 5th IEEE Conference on Open Architectures and Network Programming (OPENARCH)*, pages 167–178. IEEE, 2002. doi:10.1109/OPNARC.2002.1019237.

9. T-H Hubert Chan and Anupam Gupta. Small hop-diameter sparse spanners for doubling metrics. *Discrete & Computational Geometry*, 41(1):28–44, 2009.
10. Shiri Chechik, Michael Langberg, David Peleg, and Liam Roditty. Fault tolerant spanners for general graphs. *SIAM Journal on Computing*, 39(7):3403–3423, 2010. doi:10.1137/090758039.
11. Paul Chew. There are planar graphs almost as good as the complete graph. *Journal of Computer and System Sciences*, 39(2):205–219, 1989. doi:10.1016/0022-0000(89)90044-5.
12. Edith Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. *SIAM Journal on Computing*, 28(1):210–236, 1998. doi:10.1137/S0097539794261295.
13. Mirela Damian, Saurav Pandit, and Sriram Pemmaraju. Distributed spanner construction in doubling metric spaces. In *International Conference on Principles of Distributed Systems*, pages 157–171. Springer, 2006.
14. Mirela Damian, Saurav Pandit, and Sriram Pemmaraju. Local approximation schemes for topology control. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 208–217, 2006.
15. Gautam Das. The visibility graph contains a bounded-degree spanner. In *Proceedings of the 9th Canadian Conference on Computational Geometry (CCCG)*, pages 70–75, 1997. URL: https://cccg.ca/proceedings/1997/.
16. Andrew Dobson and Kostas E. Bekris. Sparse roadmap spanners for asymptotically near-optimal motion planning. *International Journal of Robotics Research*, 33(1):18–47, 2014. doi:10.1177/0278364913498292.
17. Michael Elkin. Computing almost shortest paths. *ACM Transactions on Algorithms*, 1(2):283–323, 2005. doi:10.1145/1103963.1103968.
18. Michael Elkin, Arnold Filtser, and Ofer Neiman. Distributed construction of light networks. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 483–492, 2020.
19. Michael Elkin and David Peleg. $(1+\varepsilon,\beta)$-spanner constructions for general graphs. *SIAM Journal on Computing*, 33(3):608–631, 2004. doi:10.1137/S0097539701393384.
20. Michael Elkin and Jian Zhang. Efficient algorithms for constructing $(1+\varepsilon,\beta)$-spanners in the distributed and streaming models. *Distributed Computing*, 18(5):375–385, 2006. doi:10.1007/s00446-005-0147-2.
21. David Eppstein. Spanning trees and spanners. In Jörg-Rüdiger Sack and Jorge Urrutia, editors, *Handbook of Computational Geometry*, pages 425–461. North-Holland, 2000. doi:10.1016/B978-044482537-7/50010-3.
22. David Eppstein and Siddharth Gupta. Crossing patterns in nonplanar road networks. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 1–9, 2017.
23. David Eppstein and Hadi Khodabandeh. On the edge crossings of the greedy spanner. In *37th International Symposium on Computational Geometry*, volume 12, page 37, 2021.
24. David Eppstein and Hadi Khodabandeh. Optimal spanners for unit ball graphs in doubling metrics. *arXiv preprint*, 2021. arXiv:2106.15234.
25. Mohammad Farshi and Joachim Gudmundsson. Experimental study of geometric t-spanners. *ACM Journal of Experimental Algorithmics*, 14:1.3:1–1.3:29, 2009. doi:10.1145/1498698.1564499.
26. Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. Graph distances in the streaming model: the value of space. In *Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 745–754. Society for Industrial and Applied Mathematics, 2005.
27. Arnold Filtser and Shay Solomon. The greedy spanner is existentially optimal. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 9–17, 2016.
28. Jie Gao, Leonidas J Guibas, John Hershberger, Li Zhang, and An Zhu. Geometric spanners for routing in mobile networks. *IEEE journal on selected areas in communications*, 23(1):174–185, 2005.

29 Lee-Ad Gottlieb. A light metric spanner. In *Proceedings of the 56th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 759–772. IEEE, 2015. doi:10.1109/FOCS.2015.52.

30 Joachim Gudmundsson, Christos Levcopoulos, and Giri Narasimhan. Fast greedy algorithms for constructing sparse geometric spanners. *SIAM Journal on Computing*, 31(5):1479–1500, 2002.

31 Jonathan P Jenkins, Iyad A Kanj, Ge Xia, and Fenghui Zhang. Local construction of spanners in the 3d space. *IEEE Transactions on Mobile Computing*, 11(7):1140–1150, 2012.

32 Lujun Jia, Rajmohan Rajaraman, and Christian Scheideler. On local algorithms for topology control and routing in ad hoc networks. In *Proceedings of the 15th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 220–229. ACM, 2003. doi:10.1145/777412.777447.

33 Iyad A Kanj, Ljubomir Perković, and Ge Xia. Computing lightweight spanners locally. In *International Symposium on Distributed Computing*, pages 365–378. Springer, 2008.

34 J. Mark Keil. Approximating the complete Euclidean graph. In Rolf Karlsson and Andrzej Lingas, editors, *Proceedings of the 1st Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 318 of *Lecture Notes in Computer Science*, pages 208–213. Springer, 1988. doi:10.1007/3-540-19487-8_23.

35 Fabian Kuhn, Thomas Moscibroda, and Rogert Wattenhofer. On the locality of bounded growth. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 60–68, 2005.

36 Hung Le and Shay Solomon. Truly optimal Euclidean spanners. In *Proceedings of the 60th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1078–1100. IEEE, 2019. doi:10.1109/FOCS.2019.00069.

37 Ning Li, Jennifer C Hou, and Lui Sha. Design and analysis of an mst-based topology control algorithm. *IEEE Transactions on Wireless Communications*, 4(3):1195–1206, 2005.

38 Xiang-Yang Li, Gruia Calinescu, Peng-Jun Wan, and Yu Wang. Localized delaunay triangulation with application in ad hoc wireless networks. *IEEE Transactions on Parallel and Distributed Systems*, 14(10):1035–1047, 2003.

39 Xiang-Yang Li, Peng-Jun Wan, and Yu Wang. Power efficient and sparse spanner for wireless ad hoc networks. In *Proceedings Tenth International Conference on Computer Communications and Networks (Cat. No. 01EX495)*, pages 564–567. IEEE, 2001.

40 James D. Marble and Kostas E. Bekris. Asymptotically near-optimal planning with probabilistic roadmap spanners. *IEEE Transactions on Robotics*, 29(2):432–444, 2013. doi:10.1109/TRO.2012.2234312.

41 Giri Narasimhan and Michiel Smid. *Geometric spanner networks*. Cambridge University Press, 2007.

42 David Peleg and Liam Roditty. Localized spanner construction for ad hoc networks with variable transmission range. *ACM Transactions on Sensor Networks (TOSN)*, 7(3):1–14, 2010.

43 David Peleg and Alejandro A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989. doi:10.1002/jgt.3190130114.

44 Liam Roditty and Uri Zwick. On dynamic shortest paths problems. *Algorithmica*, 61(2):389–401, 2011. doi:10.1007/s00453-010-9401-5.

45 Johannes Schneider and Roger Wattenhofer. A log-star distributed maximal independent set algorithm for growth-bounded graphs. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 35–44, 2008.

46 Michiel Smid. The weak gap property in metric spaces of bounded doubling dimension. In *Efficient Algorithms*, pages 275–289. Springer, 2009.

47 Wenjie Wang, Cheng Jin, and Sugih Jamin. Network overlay construction under limited end-to-end reachability. In *Proceedings of the 24th Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 3, pages 2124–2134. IEEE, 2005.

A Omitted Proofs from Section 3

▶ **Lemma 5.** *The spanner returned by* Centralized-Spanner *has a stretch factor of* $1+\epsilon$.

Proof. We recall that the output of Approximate-Greedy is a light-weight bounded-degree $(1+\epsilon')$-spanner of the complete weighted graph on the point set. So an edge e of length $|e| > 1+\epsilon'$ cannot be used to approximate any edges in the UBG, i.e. if $(x,y) \in E$ then e cannot belong to the shortest path between x and y in S; otherwise the length of the path would exceed $1+\epsilon'$ which cannot happen. So we may safely remove these edges in the first step of the refinement procedure without replacing them.

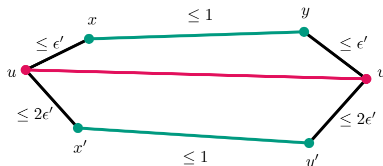


Figure 3 An edge (x,y) of the UBG that uses a longer than unit length edge (u,v) of the spanner on its shortest path, which is then replaced by (x',y') during the replacement procedure.

Also, any edge of length in the range $(1, 1+\epsilon']$ that is not used in a shortest path between any two endpoints of an edge of UBG can be removed as well, because removing them does not change the stretch-factor of the spanner. Now consider an edge $(x,y) \in E$ of the UBG that uses a spanner edge $e = (u,v) \in S$ that $|e| \in (1, 1+\epsilon']$ on its shortest path. We want to prove that after the replacement of e, the shortest path between x and y remains within $1+\epsilon$ factor of their distance. Clearly, we have $\|ux\| \leq \epsilon'$ and $\|vy\| \leq \epsilon'$; otherwise the length of the path $xuvy$ would be more than $1+\epsilon'$, contradicting the fact that it is approximating an edge of length at most 1. This shows that (x,y) would be a valid replacement edge for e. So we can safely assume that the algorithm finds a (possible weak) replacement edge $(x',y') \in E$ for e (Figure 3). This replacement edge might be a normal replacement edge or a weak replacement edge. Either way, we have $\|ux'\| \leq 2\epsilon'$ and $\|vy'\| \leq 2\epsilon'$. By the triangle inequality

$$\|x'x\| \leq \|x'u\| + \|ux\| \leq 2\epsilon' + \epsilon' = 3\epsilon'$$

Similarly, $\|yy'\| \leq 3\epsilon'$. Therefore

$$\|x'y'\| \leq \|x'x\| + \|xy\| + \|yy'\| \leq \|xy\| + 6\epsilon' \tag{1}$$

Denote the shortest spanner path between x and x' by $P_{xx'}$ and similarly define $P_{yy'}$. Consider the spanner path $P = P_{xx'}x'y'P_{y'y}$ that connects x and y. Using Equation 1 the length of the path P is

$$|P| = |P_{xx'}| + \|x'y'\| + |P_{y'y}| \leq \|xy\| + 6\epsilon' + |P_{xx'}| + |P_{y'y}| \tag{2}$$

The changes that we make in the refinement process do not affect the length of short paths like $P_{xx'}$ and $P_{y'y}$. So we have

$$|P_{xx'}| \leq (1+\epsilon')\|xx'\| \leq 3\epsilon'(1+\epsilon')$$

Similarly, $|P_{yy'}| \leq 3\epsilon'(1+\epsilon')$. Putting these into Equation 2 and using $\epsilon = 36\epsilon'$,

$$|P| \leq \|xy\| + 6\epsilon' + 6(1+\epsilon')\epsilon' \leq \|xy\| + \frac{\epsilon}{1+\epsilon'} \tag{3}$$

But since e was previously approximating the edge (x,y), we know that $(1+\epsilon')\|xy\| \geq |e| > 1$ or equivalently $\|xy\| > 1/(1+\epsilon')$. Substituting this into Equation 3,

$$|P| \leq \|xy\| + \epsilon\|xy\| = (1+\epsilon)\|xy\|$$

So S is a $(1+\epsilon)$-spanner of G. ◀

▶ **Lemma 6.** *The spanner returned by* CENTRALIZED-SPANNER *has a weight of* $\mathcal{O}(1)\mathbf{w}(MST)$.

Proof. Again, we use the fact that the output of APPROXIMATE-GREEDY has weight $\mathcal{O}(1)\mathbf{w}(MST(G))$. During the refinement process, every edge is replaced by an edge of smaller length, so the whole weight of the graph does not increase during the refinement process. Therefore in the end $\mathbf{w}(S) = \mathcal{O}(1)\mathbf{w}(MST(G))$. ◀

▶ **Lemma 7.** *The spanner returned by* CENTRALIZED-SPANNER *has bounded degree.*

Proof. It is clear from the algorithm that immediately after processing an edge $e = (u,v)$, the degree of u and v does not increase; it may decrease due to the removal of the edge which is fine. But if a replacement edge (x,y) is added after the removal of e then the degree of x and y is increased by at most 1. We need to make sure this increment is bounded for every vertex.

Let x be an arbitrary vertex of G and let (x,y) and (x,z) be two replacement edges that have been added to x in this order as a result of the refinement process. We claim that $\|yz\| > \epsilon'$ holds. Assume, on the contrary, that $\|yz\| \leq \epsilon'$, and also assume that (x,z) has been added in order to replace an edge (u,v) of the spanner. Then by the triangle inequality

$$\|vy\| \leq \|vz\| + \|zy\| \leq 2\epsilon'$$

Also $\|ux\| \leq \epsilon' < 2\epsilon'$ because (x,z) is added to replace (u,v). But the last two inequalities contradict the fact that (x,y) cannot be a weak replacement for (u,v).

Now that we have proved $\|yz\| > \epsilon'$ we can use the packing property of the bounded doubling dimension to bound the number of such replacement edges around x. All the other endpoints of such replacement edges are included in ball of radius 1 around x, and the distance between every two such points is at least ϵ'. Thus by the packing property there can be at most $(\frac{4}{\epsilon'})^d = \epsilon^{-\mathcal{O}(d)}$ many replacement edges incident to x. ◀

B Omitted Proofs from Section 4

▶ **Lemma 9.** DISTRIBUTED-SPANNER *can be done in* $\mathcal{O}(\log^* n)$ *rounds of communication.*

Proof. The pre-processing step of finding the maximal independent set takes $\mathcal{O}(\log^* n)$ rounds of communication [35]. Retrieving the 2-hop neighborhood information can be done in $\mathcal{O}(1)$ rounds of communication. Computing the greedy spanner is done locally, and the edges are sent to their endpoints, which again can be done in $\mathcal{O}(1)$ rounds of communication. Overall, the algorithm requires $\mathcal{O}(\log^* n)$ rounds of communication. ◀

▶ **Lemma 10.** *The spanner returned by* DISTRIBUTED-SPANNER *has a stretch factor of* $1+\epsilon$.

Proof. From section 3 we know that $\mathcal{S}_w$ is a light-weight bounded-degree $(1+\epsilon)$-spanner of $\mathcal{N}^2(w)$. Let $u, v \in V$ be chosen arbitrarily. We need to make sure there is a path of length at most $(1+\epsilon)d_G(u,v)$ between u and v in the output.

First we prove this for the case that $(u,v) \in E$. So let $e = (u,v) \in E$. Then u is either in I or has a neighbor in I, according to choice of I. In any case, the edge e belongs to $\mathcal{N}^2(w)$ for some $w \in I$, which means that there is a path $P \subset \mathcal{S}_w$ of length at most $(1+\epsilon)|e|$

that connects u and v. The edges of P are all included in the final spanner according to the algorithm, so the output includes this path between u and v, which has a length at most $(1+\epsilon)|e|$ and so the distance inequality is satisfied.

If $(u,v) \notin E$, we can take the shortest path $u = p_0, p_1, \cdots, p_k = v$ between them in G and append the corresponding $(1+\epsilon)$-approximate paths $P_0, P_1, \cdots, P_{k-1}$ of the edges $p_0p_1, p_1p_2, \cdots, p_{k-1}p_k$, respectively, to get a $(1+\epsilon)$-approximate path for $p_0p_1 \cdots p_k$. This implies that the stretch factor of the output is indeed $1+\epsilon$. ◀

▶ **Lemma 11.** *The spanner returned by* Distributed-Spanner *has a bounded degree.*

Proof. First we use the packing lemma to prove that any vertex $v \in V$ appears at most a constant number of times in different neighborhoods, $\mathcal{N}^2(w)$ for $w \in I$. Because $v \in \mathcal{N}^2(w)$ implies that $\|vw\| \leq 2$, any vertex $w \in I$ such that $v \in \mathcal{N}^2(w)$ should be contained in the ball of radius 2 around v. But all such ws are chosen from I, which is an independent set of G, so the distance between every two such vertex is at least 1. By the packing property, the maximum number of such vertices would be $8^d = \mathcal{O}(1)$.

Now that every vertex appears in at most in 8^d different sets $\mathcal{N}^2(w)$, for $w \in I$, and from section 3 we already knew that every vertex has bounded degree in any of $\mathcal{S}_w$s, it immediately follows that every vertex has bounded degree in the final spanner. ◀

▶ **Lemma 13.** *Define $\mathcal{T}$ to be the optimal Steiner tree on the set of vertices $\mathcal{N}^3(w)$, where only vertices in $\mathcal{N}^2(w)$ are required and the rest of them are optional. Then*

$$\mathbf{w}(MST(\mathcal{N}^2(w))) \leq 2\mathbf{w}(\mathcal{T})$$

Proof. This is a well-known fact that implies a 2-approximation for minimum Steiner tree problem. The idea is if we run a full DFS on the vertices of $\mathcal{T}$ and we write every vertex once we open and once we close it, then we get a cycle whose shortcut on optional edges will form a path on the required vertices. The weight of the cycle is at least $\mathbf{w}(MST(\mathcal{N}^2(w)))$ and at most $2\mathbf{w}(\mathcal{T})$, which proves the result. ◀

▶ **Lemma 14.** *Let $\mathcal{S}^*$ be the output of* Centralized-Spanner*(G, ϵ) and let $\mathcal{S}^*_w$ be the induced subgraph of $\mathcal{S}^*$ on $\mathcal{N}^3(w)$. Then*

$$\mathbf{w}(\mathcal{T}) \leq \mathbf{w}(\mathcal{S}^*_w)$$

Proof. We prove that for $\epsilon \leq 1$, $\mathcal{S}^*_w$ forms a forest that connects all the vertices in $\mathcal{N}^2(w)$ in a single component. So $\mathcal{S}^*_w$ is a feasible solution to the minimum Steiner tree problem on the set of vertices $\mathcal{N}^3(w)$ with required vertices being $\mathcal{N}^2(w)$. Thus $\mathbf{w}(\mathcal{T}) \leq \mathbf{w}(\mathcal{S}^*_w)$.

Now we just need to prove that the vertices in $\mathcal{N}^2(w)$ are connected in $\mathcal{S}^*_w$. Let u be an i-hop neighbor of w and v be an $i+1$-hop neighbor of w for some $w \in I$ and $i = 0, 1$. Assume that $(u,v) \in E$. It is enough to prove that u and v are connected in $\mathcal{S}^*_w$. In order to do so, we observe that there is a path of length at most $(1+\epsilon)\|uv\|$ between u and v in $\mathcal{S}^*$. We show that this path is contained in $\mathcal{N}^3(w)$ and we complete the proof in this way, because $\mathbf{w}(\mathcal{S}^*_w)$ is nothing but the induced subgraph of $\mathcal{S}^*$ on $\mathcal{N}^3(w)$.

Assume, on the contrary, that there is a vertex $x \notin \mathcal{N}^3(w)$ on the $(1+\epsilon)$-path between u and v. This means that x is not a 1-hop neighbor of any of u and v, because otherwise x would have been in $\mathcal{N}^3(w)$. So $\|ux\| > 1$ and $\|vx\| > 1$. Thus the length of the path would be at least $\|ux\| + \|xv\| > 2 \geq (1+\epsilon) \geq (1+\epsilon)\|uv\|$ which is a contradiction. ◀

C Omitted Proofs from Section 5

▶ **Lemma 17.** *The union of the balls of radius* $1/2$ *around the centers in* $\mathcal{I}(w)$ *cover* $\mathcal{N}^2(w)$*. Furthermore, the size of* $\mathcal{I}(w)$ *is bounded by a constant.*

Proof. Let $v \in \mathcal{N}^2(w)$ be an arbitrary point. Thus there exists $u \in \mathcal{N}^1(w)$ that $v \in \mathcal{N}^1(u)$. Let $I_{1/4}(u)$ be the maximal independent set of the vertices $\mathcal{N}^1(u)$ in $G_{\leq 1/4}$, that u calculates and sends to w in the first step. There exists $v' \in I_{1/4}(u)$ that $\|vv'\| \leq 1/4$. Similarly, there exists $v'' \in \mathcal{I}(w)$ that $\|v'v''\| \leq 1/4$. By the triangle inequality, $\|vv''\| \leq 1/2$, i.e. v is covered by a ball of radius $1/2$ around v''.

On the other hand, $\mathcal{I}(w)$ is contained in a ball of radius 2 and every pair of points in $\mathcal{I}(w)$ have a distance of at least $1/4$. Thus, by the packing lemma, he size of $\mathcal{I}(w)$ is bounded by a constant. ◀

▶ **Lemma 18.** *The union of the spanners* $\mathcal{S}_{\leq 1/2}(v)$ *for* $v \in \mathcal{I}(w)$ *is a* $(1+\epsilon)$*-spanner of* $\mathcal{N}^2(w)$ *in* $G_{\leq 1/2}$*. The maximum degree and the lightness of this spanner are both bounded by constants.*

Proof. First, we prove the $1+\epsilon$ stretch-factor. Let (u, v) be a pair in $\mathcal{N}^2(w)$ such that $\|uv\| \leq 1/2$. By Lemma 17 we know there exists $u' \in \mathcal{I}(w)$ that $\|uu'\| \leq 1/2$. Thus $\|vu'\| \leq 1$ which means that $u, v \in \mathcal{N}^1(u')$ and there would be a $(1+\epsilon)$-path for this pair in $\mathcal{S}_{\leq 1/2}(u')$, which would be present in the union of the spanners.

The degree bound follows from the fact that, by the packing lemma, every point in $\mathcal{N}^2(w)$ is appeared in at most a constant number of one-hop neighborhoods and therefore in at most a constant number of spanners constructed the elements in $\mathcal{I}(w)$. Since in every spanner it has a bounded degree, in the union it will have a bounded degree as well.

To prove the lightness bound we follow a similar approach to the proof of Proposition 15. The weight of each spanner $\mathcal{S}_{\leq 1/2}(v)$ is $\mathcal{O}(1)\mathbf{w}(MST(\mathcal{N}^1(v)))$. The weight of the MST is at most twice the weight of the optimal Steiner tree on $\mathcal{N}^2(v)$ with the required vertices being $\mathcal{N}^1(v)$. And the weight of this optimal Steiner tree is at most equal to the weight of the induced sub-graph of an (asymptotically) optimal $(1+\epsilon)$-spanner of G on the subset of vertices $\mathcal{N}^2(v)$. Summing up these subgraphs for different vs and different ws would end up with adding at most a constant factor to the weight of the optimal spanner, which proves that the lightness would be bounded by a constant. ◀

▶ **Lemma 20.** *The union of the spanners* $\mathcal{S}'(w)$ *and* $\mathcal{S}''(v)$ *for* $v \in \mathcal{I}'(w)$ *forms a* $(1+\epsilon)$*-spanner of* $\mathcal{N}^2(w)$ *in* $G_{>1/2}$*. The maximum degree and the lightness of this spanner are btoh bounded by constants.*

Proof. Again, we first prove the $1+\epsilon$ stretch-factor of the spanner. Let (u, v) be a pair in $\mathcal{N}^2(w)$ that $\|uv\| > 1/2$. Let u' and v' be centers in $\mathcal{I}'(w)$ that are at distance of at most $\epsilon/20$ from u and v, respectively. Such centers exist according to Lemma 19. Consider the $(1+\epsilon)$-path connecting u to u' in $\mathcal{S}''(u')$ and the $(1+\epsilon)$-path connecting v to v' in $\mathcal{S}''(v')$. We can attach these paths together with the $(1+\epsilon/5)$-path between u' and v' in $\mathcal{S}'(w)$ to get a path between u and v. The stretch of this path would be at most

$$\frac{(1+\epsilon)(\|uu'\| + \|vv'\|) + (1+\epsilon/5)\|u'v'\|}{\|uv\|} = \frac{(1+\epsilon)(\|uu'\| + \|vv'\|)}{\|uv\|} + \frac{(1+\epsilon/5)\|u'v'\|}{\|uv\|}$$

But,

$$\frac{(1+\epsilon)(\|uu'\| + \|vv'\|)}{\|uv\|} \leq \frac{(1+\epsilon)\epsilon/10}{1/2} \leq \frac{2\epsilon}{5}$$

Also,

$$\frac{(1+\epsilon/5)\|u'v'\|}{\|uv\|} \leq \frac{(1+\epsilon/5)(\|uv\|+\|uu'\|+\|vv'\|)}{\|uv\|} \leq 1+\epsilon/5+\frac{(1+\epsilon/5)\epsilon/10}{\|uv\|}$$

Bounding the last term,

$$\frac{(1+\epsilon/5)\epsilon/10}{\|uv\|} \leq \frac{(6/5)\epsilon/10}{1/2} = \frac{6\epsilon}{25}$$

Therefore, the stretch of the uv-path would be upper bounded by,

$$\frac{2\epsilon}{5}+1+\epsilon/5+\frac{6\epsilon}{25} < 1+\epsilon$$

An approach similar to the proof of Lemma 18 shows that the degree of every vertex in the union of $\mathcal{S}''(v)$s would be bounded by a constant. We do not repeat the details of the proof here. From the properties of our centralized construction, the degree of every vertex would be bounded in $\mathcal{S}'(w)$ as well. Thus, the degree of every vertex in the union of these spanners would be bounded by a constant.

To prove the lightness bound, we bound the weight of each spanner separately. First, we bound the total weight of $\mathcal{S}'(w)$. We know from the properties of our centralized construction, that $\mathbf{w}(\mathcal{S}'(w)) = \mathcal{O}(1)\mathbf{w}(MST(\mathcal{I}'(w)))$. But $\mathbf{w}(MST(\mathcal{I}'(w))) \leq 2MST(\mathcal{N}^2(w))$, so $\mathbf{w}(\mathcal{S}'(w)) = \mathcal{O}(1)\mathbf{w}(MST(\mathcal{N}^2(w)))$. Therefore, by Lemma 13 and Lemma 14 the total weight of $\mathcal{S}'(w)$ spanners for different centers w would sum up to at most a constant factor of the weight of the optimal spanner.

Next, we bound the total weight of $\mathcal{S}''(v)$ spanners. Again, we know from the properties of our centralized construction that $\mathbf{w}(\mathcal{S}''(v)) = \mathcal{O}(1)\mathbf{w}(MST(\mathcal{N}^{\epsilon/20}(v)))$. Assuming that $\mathcal{S}^*$ is an optimal spanner on the point set, we can observe that any $(1+\epsilon)$-path (in $\mathcal{S}^*$) between any pair of vertices in $\mathcal{N}^{\epsilon/20}(v)$ must be completely contained in a ball of radius $3\epsilon/20$, otherwise the length of the path would be more than $(1+\epsilon)\epsilon/10$, the maximum allowed length for any $(1+\epsilon)$-path of any pair in the $\mathcal{N}^{\epsilon/20}(v)$ neighborhood. Therefore, the induced sub-graph of $\mathcal{S}^*$ on $\mathcal{N}^{3\epsilon/20}(v)$ has a connected component connecting the vertices of $\mathcal{N}^{\epsilon/20}(v)$. Thus, its weight is at least equal to the weight of a minimum Steiner tree on $\mathcal{N}^{3\epsilon/20}(v)$, with the required vertices being $\mathcal{N}^{\epsilon/20}(v)$. This is at least equal to $\mathbf{w}(MST(\mathcal{N}^{\epsilon/20}(v)))/2$. Therefore, the weight of $\mathcal{S}''(v)$ is bounded above by a constant factor of the weight of the induced sub-graph of $\mathcal{S}^*$ on $\mathcal{N}^{3\epsilon/20}(v)$. Summing up these bounds for every v in every w would lead to at most a constant repetitions of every vertex and every edge (similar to Proposition 15) in $\mathcal{S}^*$, which shows that the total weight of $\mathcal{S}''(v)$ for different vertices of v would be bounded by a constant factor of the weight of the optimal spanner. ◀

Improved Deterministic Connectivity in Massively Parallel Computation

Manuela Fischer
ETH Zürich, Switzerland

Jeff Giliberti
ETH Zürich, Switzerland

Christoph Grunau
ETH Zürich, Switzerland

Abstract

A long line of research about connectivity in the Massively Parallel Computation model has culminated in the seminal works of Andoni et al. [FOCS'18] and Behnezhad et al. [FOCS'19]. They provide a randomized algorithm for low-space MPC with conjectured to be optimal round complexity $O(\log D + \log\log_{\frac{m}{n}} n)$ and $O(m)$ space, for graphs on n vertices with m edges and diameter D. Surprisingly, a recent result of Coy and Czumaj [STOC'22] shows how to achieve the same deterministically. Unfortunately, however, their algorithm suffers from large local computation time.

We present a deterministic connectivity algorithm that matches all the parameters of the randomized algorithm and, in addition, significantly reduces the local computation time to nearly linear.

Our derandomization method is based on reducing the amount of randomness needed to allow for a simpler efficient search. While similar randomness reduction approaches have been used before, our result is not only strikingly simpler, but it is the first to have efficient local computation. This is why we believe it to serve as a starting point for the systematic development of computation-efficient derandomization approaches in low-memory MPC.

2012 ACM Subject Classification Theory of computation → Massively parallel algorithms; Theory of computation → MapReduce algorithms

Keywords and phrases Massively Parallel Computation, MPC, MapReduce, Deterministic Algorithms, Connectivity, Hitting Set, Maximum Matching, Derandomization

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.22

Funding *Christoph Grunau*: Supported by the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement No. 853109).

1 Introduction

Due to the ever-increasing amount of data available, memory has grown to become a major bottleneck, which makes many traditional graph algorithms inefficient or even inapplicable. To overcome this obstacle, inspired by the MapReduce paradigm [17], several computation frameworks for large-scale graph processing across multiple machines have been proposed. The Massively Parallel Computation (MPC) model is a clean, theoretical abstraction of these frameworks and thus serves as a basis for the systematic study of memory-restricted distributed algorithms. Introduced by Karloff et al. [27] and Feldman et al. [19] in 2010, it was later refined in a sequence of works and has become tremendously popular over the past decade.

© Manuela Fischer, Jeff Giliberti, and Christoph Grunau;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 22; pp. 22:1–22:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

MPC Model

In the MPC model, the distributed network consists of $\mathbf{M}$ machines, having local memory $\mathbf{S}$ each. The input is distributed across the machines and the computation proceeds in synchronous rounds. In each round, each machine performs an arbitrary *local computation* and then communicates up to $\mathbf{S}$ data. All messages sent and received by each machine in each round have to fit into the machine's local space. The main complexity measure of an algorithm is its *round complexity*, that is, the number of rounds needed by the algorithm to solve the problem. Secondary complexity measures of an algorithm are its *global memory* usage – i.e., the number of machines times the memory per machine required – as well as the *total computation* performed by machines to run the algorithm, i.e., the (asymptotic) sum of the local computation performed by each machine.

We focus on the design of fully scalable graph algorithms in the *low-memory* MPC model, where each machine has strongly sublinear memory. More precisely, an input graph $G = (V, E)$, with n vertices and m edges, is distributed arbitrarily across machines with local memory $\mathbf{S} = O(n^\delta)$ each, for some constant $0 < \delta < 1$, so that the global space is $\mathbf{S}_{Global} = \Omega(n + m)$.

Graph Algorithms and Connectivity

In this model, fundamental graph and optimization problems have recently gained a lot of attention. There is a plethora of work on the problems of connectivity, matching, maximal independent set, vertex cover, coloring, and many more (see, e.g., [6, 20, 12, 15, 7, 23, 16]).

One particularly important (and arguably the most central) graph problem that has received increasing attention over the past few years is the one of connectivity. This is not only a problem of independent interest, but it serves as a subroutine for many algorithms.

▶ **Definition 1** (Connectivity Problem). *Let $G = (V, E)$ be an undirected graph. The goal is to compute a function $cc\colon V \to \mathbb{N}$ such that every vertex $u \in V$ knows $cc(u)$ and for any pair of vertices $u, v \in V$, u and v are connected in G if and only if $cc(u) = cc(v)$.*

A sequence of works [3, 4, 29, 7, 33, 10, 7] on this problem culminated in a randomized algorithm by Behnezhad et al. [7] that finds all connected components of a graph with diameter D in $O(\log D + \log \log_{\frac{m}{n}} n)$ rounds.

In a very recent breakthrough, Coy and Czumaj [12] obtained the same round complexity with a deterministic algorithm. Their derandomization approach, however, comes at a cost of heavy local computation, which makes it impractical for large-scale applications.

Deterministic Algorithms and Derandomization

While the problem of connectivity is of independent interest, it is instructive to view the above results in a broader context of deterministic algorithms and derandomization.

Notably, for almost a decade, (almost) all the research in the domain of Massively Parallel Computation has focused on the study of randomized algorithms. Only recently, a sequence of works has aimed at exploring the power of the (low-memory) MPC model restricted to deterministic algorithms [5, 16, 13, 15, 12]. They demonstrate that several graph problems can be solved deterministically with (asymptotic) complexity bounds that are comparable to those of the randomized algorithms. The main ingredients of these results are derandomization methods specifically tailored to the low-memory MPC model: they are designed to cope with the limited memory per machine while exploiting the power of *local computation* and *all-to-all communication* in this setting.

This quest for efficient derandomization techniques has become one of the main problems of the area. Unfortunately, current derandomization frameworks suffer from long local running time (e.g., large polynomial or even exponential in n^{δ}). In fact, as noted in [16], allowing heavy local computation might provide an advantage in the context of distributed and parallel derandomization. However, especially in performance-oriented scenarios, local computation may quickly become a critical parameter. It thus emerges as a natural direction to study deterministic algorithms whose total computation matches that of their randomized counterparts.

1.1 Our Contribution

We address this issue by presenting the first computation-efficient deterministic algorithm for the problem of graph connectivity in the strongly sublinear memory regime of MPC.

▶ **Theorem 2** (Deterministic Connectivity). *There is a strongly sublinear MPC algorithm that given a graph with diameter D, identifies its connected components in $O(\log D + \log\log_{\frac{m}{n}} n)$ rounds deterministically using $O(n+m)$ global space and $\tilde{O}(m)$ total computation.*

The total computation of our algorithm significantly improves over the poly(n)-bound of Coy and Czumaj [12], with no loss in the round complexity. In fact, our algorithm matches even the state-of-the-art randomized algorithm [7] in all parameters up to a polylogarithmic factor in the local running time.

While the connectivity algorithm is of independent interest, our result provides a number of other qualitative advantages. For instance, our analysis relies only on pairwise independence as opposed to the almost $O(\log n)$-wise independence of [12]. Moreover, to the best of our knowledge, our result is the first that uses the framework of limited independence for derandomization without incurring a significant loss in one of the parameters (e.g., in the total computation time), and hence may be of practical interest. Furthermore, due to their simplicity, our analyses may serve as a friendly introduction to deterministic algorithms via the framework of bounded independence and, hopefully, as a stepping stone to the more systematic development of computation-efficient derandomization.

1.2 Randomized Connectivity Algorithms in a Nutshell

We present the intuition of the randomized connectivity algorithms by Andoni et al. [3] and Behnezhad et al. [7]. For a broader overview of connectivity algorithms, see Section 1.4.

Vertex Contraction

The main idea behind connectivity algorithms working in $\tilde{O}(\log D)$ rounds is to repeatedly perform *vertex contractions* [3]. Contracting (often also called *relabeling*) a vertex u to an adjacent vertex v means deleting the edge $\{u, v\}$ and connecting v to all the vertices adjacent to u. The simplest way to implement this contraction-based approach is to first appoint a random subset of the vertices as *leaders* (by letting each vertex independently with probability $\frac{1}{2}$ become a leader), and then to contract non-leader vertices to one of their leader neighbors (if any). This approach requires $O(\log n)$ rounds with high probability.

Vertex Contraction with Levels and Budgets (Andoni et al. [3])

A crucial observation to speed up the vertex contractions – going back to the graph exponentiation approach by Lenzen and Wattenhofer [31] – is to let each vertex expand its neighborhood to neighbors of neighbors by adding new edges (without changing the connectivity). In fact, if every vertex reaches degree $\Omega(d)$ by expanding its neighborhood in

$O(\log D)$ rounds, we can mark vertices to be a leader with probability $\approx \frac{\log n}{d}$. As a result, each non-leader vertex has a leader in its neighborhood and the number of remaining vertices is $\tilde{O}(\frac{n}{d})$.

In their algorithm, Andoni et al. [3] assign a level to every vertex which has not been contracted yet. Vertices at level i have a budget of b_i for expanding their neighborhood, i.e., each vertex at level i can add at most b_i neighbors. The initial budget b_0 is set to $\min(n^{\delta/2}, \sqrt{\frac{m}{n}})$ to maintain global space $O(m)$. At iteration i, every vertex either increases its degree to b_i or finds its connected component. As explained above, we thus can mark leader vertices with probability $\frac{\log n}{b_i}$ and perform contractions to reduce the problem size to $\tilde{O}(n/b_i)$. Hence, the budgets of remaining vertices can be updated to $b_{i+1} = b_i^{1+c}$, for a small constant c, while using the same global space. Overall, after $O(\log\log_{\frac{m}{n}} n)$ iterations, there will be a unique vertex left in each connected component.

Random Leader Contraction (Behnezhad et al. [7])

To further improve the round complexity, Behnezhad et al. [7] design an algorithm that applies vertex contractions and increases the budgets of vertices in an asynchronous manner, e.g., at a given time two active vertices can have different budgets. In each round, their algorithm (informally) ensures that each vertex either learns its 2-hop neighborhood or increases its budget. We here focus on the routine that defines the budgets' increase, as this is the only step involving randomness.

Consider the subgraph induced by vertices with budget level i. The crucial observation is that if a vertex has $\Omega(b_i)$ many neighbors of the same level, then contracting all of them allows us to recuperate $\Omega(b_i^2)$ budget. If each vertex is elected as a leader with probability $\approx \frac{\log n}{b_i}$, and non-leader vertices contracted to an arbitrary neighboring leader, then leaders can increase their level without exceeding the total memory.

Increasing Initial Budget using Matching (Behnezhad et al. [7])

To allow each vertex to start with a $\text{poly} \log n$ budget, a randomized constant-round algorithm (see [7, Algorithm 3]) reduces the number of vertices of G by a constant factor. By running it for $O(\log\log n)$ MPC rounds, the problem size decreases from n to $n/\text{poly}\log n$. Intuitively, this algorithm works by contracting a constant fraction of the vertices to their lowest-ID neighbors as follows. Each vertex proposes to be contracted to its neighbor with smallest ID. A deterministic conflict resolving phase results in a graph of size $\Omega(n)$ consisting of *vertex-disjoint paths*. Contracting along the edges of a constant-approximate *maximum matching* in this graph with maximum degree 2 thus allows to contract $\Omega(n)$ vertices as desired.

1.3 Deterministic Connectivity: Comparison with the State-of-the-Art

We next present the main ideas behind the recent deterministic connectivity algorithm of Coy and Czumaj [12].

Coy and Czumaj [12] identify and extract the only two sources of randomization from the algorithms of [3, 7], namely matching and hitting set. On the one hand, as outlined in Section 1.2, a constant approximation of matching in graphs with maximum degree 2 can be used for the initial budget increase. On the other hand, the random leader contraction can be formulated as a variant of set cover, which we refer to as hitting set with all sets of the same size (see Definition 9 for a precise definition).

As these are the only steps involving randomness (as outlined in Section 1.2), the (efficient) derandomization of these two constant-round key algorithmic primitives immediately leads to an (efficient) deterministic connectivity algorithm. In fact, their derandomization together with the $O(\log D + \log\log n)$ randomized algorithm due to Behnezhad et al. [7] results in the state-of-the-art deterministic connectivity algorithm in low-memory MPC [12].

Interestingly, because of the conditional lower bound framework (conditioned on the widely believed 1-vs-2-cycles conjecture for low-space MPC algorithms) due to Ghaffari et al. [22] and its extension to the deterministic setting due to Czumaj et al. [14], the two underlying problems of matching and hitting set do not admit any *component-stable*[1] constant-round deterministic algorithm. Hence, the authors in [12] incorporate in their work derandomization techniques that are highly non-component-stable.

While their adopted derandomization framework is well-established, its efficient implementation for obtaining a deterministic connectivity algorithm on an MPC with low local space and optimal global space requires to overcome several challenges. Although the algorithm from [12] achieves optimal space guarantees, the computation is suboptimal for both derandomization steps. We refine these to obtain a more efficient deterministic connectivity algorithm, as explained next.

Maximum Matching

In [12], the problem of approximating maximum matching in graphs of maximum degree at most two is solved by searching the space of a randomized process based on pairwise independent hash functions, which are specified by $(2\log n + O(1))$ random bits. As each of the $O(n^2)$ hash functions is evaluated $O(n)$ times, with each evaluation taking $\text{poly}\log n$ time, the resulting total computation is $\tilde{O}(n^3)$. We reduce the *seed length*, i.e., the total number of random bits needed, to $O(\log\log n)$ and, as a result, obtain $\tilde{O}(n)$ total computation.

Hitting Set

For a hitting set instance with n elements and a collection of n subsets of size b, the algorithm from [12] finds a hitting set of size $O(nb^{-1/5})$ by derandomizing a simple random sampling approach based on a $O(\log_b(n))$-wise $1/\text{poly}(n)$-approximately independent family of hash functions of size $\text{poly}(n)$. The distributed implementation of the method of conditional expectation for this process takes global space $O(nb)$ and $\text{poly}(n)$ total computation.

We provide a low-memory MPC algorithm that solves the same hitting set instance using only pairwise independent random choices with $n \cdot \text{poly}(b)$ global space and $n \cdot \text{poly}(b)$ total computation. Thus, the dependency on n improves polynomially when $b \ll n$. It turns out that using this hitting set algorithm as a subroutine in our connectivity algorithm allows us to obtain an algorithm with total computation $\tilde{O}(m)$. We also note that several other works [9, 21, 39] solve the hitting set problem deterministically in the context of graph spanners in CONGEST and CONGESTED-CLIQUE using similar derandomization techniques. However, these are not straightforward to implement in the low-memory MPC model.

Finally, it is worth observing that because of the shorter seeds, the MPC implementation of both matching and hitting set algorithms is significantly simplified as we can perform a simple brute force search instead of using the method of conditional expectation.

[1] The notion of component-stability intuitively refers to the property that the choices of any vertex over the course of the algorithm are affected only by vertices in its same connected component.

1.4 Further Related Work

The connectivity problem in low-memory MPC was studied by Andoni et al. [3] who presented an $O(\log D \cdot \log\log_{\frac{m}{n}} n)$ randomized algorithm, which improves upon the classic $O(\log n)$ bound derived from earlier works in the PRAM model. Concurrently, for graphs with large spectral gap λ, i.e., $\Omega(1/\text{poly}\log(n))$, the bound was improved in [4] developing a randomized $O(\log\log n + \log(1/\lambda))$ algorithm. Then, a near-optimal parallel randomized algorithm that in $O(\log D + \log\log_{\frac{m}{n}} n)$ rounds determines all connected components was developed by Behnezhad et al. [7]. Subsequently, Liu et al. [33] extended the same result to the arbitrary CRCW PRAM model, which is less computationally powerful than MPC, achieving such result with good probability[2]. Moreover, by developing a method that converts randomized PRAM algorithms to highly randomness-efficient MPC algorithms, Charikar et al. [10] achieved a super-polynomial saving in the randomness used in [7], showing that $(\log n)^{O(\log D + \log\log_{m/n} n)}$ random bits suffice (with good probability), provided that the global space is $\Omega((n+m)\cdot n^{\delta})$. The current deterministic state-of-the-art algorithm for connectivity is due to Coy and Czumaj [12] who obtained a deterministic $O(\log D + \log\log_{\frac{m}{n}} n)$ algorithm with asymptotically optimal space.

Finally, let us note that the connectivity problem has been studied in other regimes as well. Lattanzi et al. [30] gave a constant-round MPC connectivity algorithm in the superlinear regime, i.e., each machine has local space $\Omega(n^{1+\delta})$. By well-known connections between linear memory MPC and the CONGESTED-CLIQUE model, [26] yields a $O(1)$-rounds randomized connectivity MPC algorithm with optimal global space. Then, Nowicki [38] showed that the same problem can be solved deterministically in $O(1)$ MPC rounds with the same memory guarantees.

On the hardness side, one of the most outstanding problems for low-space MPCcomplexity is the problem of distinguishing whether an input graph is an n-vertex cycle or consists of two $\frac{n}{2}$-vertex cycles (see, e.g., [41, 37] for more information). Based on the conjectured $\Omega(\log n)$ low-memory MPC round-complexity lower bound for the 1-vs-2-cycles problem, Behnezhad et al. [7] show an $\Omega(\log D)$ lower bound for computing connected components in general graphs with diameter $D \geq \log^{1+\Omega(1)} n$. Coy and Czumaj in [12] extend the same conditional lower bound to the entire spectrum of D proving that no connectivity algorithm can achieve $o(\log D)$ MPC round complexity.

2 Preliminaries

2.1 Primitives in Low-Space MPC

There are a number of well-known MPC primitives that will be used as black-box tools. These have been studied in the MapReduce framework and can be implemented in the MPC model with stricly sublinear space per machine and linear global space. We will use the following lemma to refer to them:

▶ **Lemma 3** ([25, 24]). *For any positive constant δ, sorting, filtering, prefix sum, predecessor, duplicate removal, and colored summation task [3] on a sequence of n tuples can be performed deterministically in MapReduce (and therefore in the MPC model) in a constant number of rounds using $\mathbf{S} = n^{\delta}$ space per machine, $O(n)$ global space, and $\tilde{O}(n)$ total computation.*

[2] with success probability at least $1 - 1/\text{poly}((m\log n)/n)$

[3] Given a sequence of n pairs of numbers $\langle color_i, x_i\rangle, i \in [n]$, with $C = \{color_i \mid i \in [n]\}$, compute $S_c = \sum_{i:color_i = c} x_i$ for all $c \in C$. Note that this problem can be easily solved by a constant sequence of map, shuffle, and reduce steps with $\langle color_i, x_i\rangle$ as key-value pairs.

Finally, observe that these basic primitives allow us to perform all of the basic computations on graphs deterministically that we will need in a constant number of MPC rounds. This includes the tasks of computing the degree of every vertex, ensuring neighborhoods of all vertices are stored on contiguous blocks of machines, sums of values among a vertex' neighborhood, and collecting the 2-hop neighborhoods provided that they fit in the memory of a single machine.

2.2 Derandomization Framework

In this section, we give an overview of the common derandomization techniques used in all-to-all communication models [9, 34] with a focus on deterministic algorithms in the strongly sublinear memory regime of MPC. A systematic introduction to the framework of limited independence can be found for example in [40, 36, 2, 35, 8, 42].

The first step is to obtain a *randomized process* that produces good results in expectation based on a small search space (i.e., short random seed) by using random variables with some limited independence. We will use a k-wise independent family of hash functions, which is defined as follows:

▶ **Definition 4** (k-wise independence)**.** *Let $N, k, \ell \in \mathbb{N}$ with $k \leq N$. A family of hash functions $\mathcal{H} = \{h : [N] \to \{0,1\}^\ell\}$ is k-wise independent if for all $I \subseteq \{1, \ldots, n\}$ with $|I| \leq k$, the random variables $X_i := h(i)$ with $i \in I$ are independent and uniformly distributed in $\{0,1\}^\ell$ when h is chosen uniformly at random from $\mathcal{H}$. If $k = 2$ then $\mathcal{H}$ is called pairwise independent. Random variables sampled from a pairwise independent family of hash functions are called pairwise independent random variables.*

The following is a well-known result about the existence and construction of such hash families:

▶ **Lemma 5** ([1, 11, 18])**.** *For every $N, \ell, k \in \mathbb{N}$, there is a family of k-wise independent hash functions $\mathcal{H} = \{h : [N] \to \{0,1\}^\ell\}$ such that choosing a uniformly random function h from $\mathcal{H}$ takes at most $k(\ell + \log N) + O(1)$ random bits, and evaluating a function from $\mathcal{H}$ takes time $\operatorname{poly}(\ell, \log N)$ time.*

If there is a randomized algorithm, over the choice of a random hash function, that gives good results in expectation, one can derandomize it by finding the right choice of (random) bits. To achieve that, if the seed length is small, one can brute force it without incurring an overhead in the global space.

In previous works this was usually not possible due to a seed length depending on n of $\Omega(\log n)$ bits, which results in hash families of size larger than the space $\mathbf{S}$ of a single machine. Instead, they used the method of conditional expectation or probabilities. There, one divides the seed into several parts and fixes one part at a time in a way that does not decrease the conditional expectation (or probability). This can be done with global coordination. We refer the interested reader for more details of the method of conditional expectation to [12, Section 2.5, Appendix A].

2.3 Reducing The Seed Length via Coloring

The following technique plays a central role for reducing the seed length of randomized processes solving *local* graph problems. As showed in [5, 16, 15], if the outcome of a vertex depends only on the random choices of its neighbors, then k-wise independence among random variables of *adjacent* vertices is sufficient. Whenever this is the case, we can find a

mapping from vertex IDs to shorter names (colors) such that adjacent vertices are assigned different names. Linial gave a 1-round distributed coloring algorithm with $O(\Delta^2 \log(n))$ colors [32]. We here adapt a more explicit 1-round distributed coloring algorithm with $O(\Delta^2 \log^2_\Delta(n))$ colors by Kuhn [28] to the MPC model, which leads to the following lemma:

▶ **Lemma 6.** *Let $G = (V, E)$ be a graph of maximum degree $\Delta \leq n^\delta$. There exists a deterministic algorithm which computes an $O(\Delta^2 \log^2_\Delta n)$ coloring of G in $O(1)$ MPC rounds using $O(n^\delta)$ local space, $O(n \cdot \text{poly}(\Delta))$ global space, and $\tilde{O}(n \cdot \text{poly}(\Delta))$ total computation.*

Proof. We start by recalling the high-level idea and then we give an efficient MPC implementation. We assume that each vertex in G is given a unique ID between 1 and n. Let p be a prime with $10\Delta \log_\Delta(n) \leq p \leq 20\Delta \log_\Delta(n)$. It is well known that such a prime always exist. Moreover, let $d = \lceil \log_\Delta(n) \rceil$. There exists $p^{d+1} \geq n$ distinct polynomials of degree at most d over $\mathbb{F}_p$. We denote by f_i the i-th such polynomial. Each color corresponds to a tuple over $\mathbb{F}_p$. Note that there are $p^2 = O(\Delta^2 \log^2_\Delta n)$ such tuples.

Let $C_i = \{(x, f_i(x)) : x \in \mathbb{F}_p\}$. Using $\Delta d < p$ together with the fact that a non-zero polynomial of degree d can have at most d zeros implies that each vertex can choose a color $c(i) \in C_i$ such that $c(i) \notin C_j$ for every neighbor j. Now, assigning each vertex i the color $c(i)$ results in a valid coloring. It remains to discuss the MPC implementation. By using the basic primitives of Lemma 3 and the assumption that $\Delta \leq n^\delta$, we can assume that the machine responsible to compute the coloring of the i-th vertex also stores the IDs of all the neighbors of i. Note that a given polynomial can be evaluated in time $\text{poly}(\log n, \Delta)$. Computing the color $c(i)$ boils down to $O(\Delta \cdot p^2) = \text{poly}(\log n, \Delta)$ polynomial evaluations. Hence, the total computation time is $\tilde{O}(n \cdot \text{poly}(\Delta))$, as desired. ◀

3 Constant Approximation of Maximum Matching

The first algorithmic step for the derandomization of the connectivity algorithm from [7] consists of solving approximate maximum matching in graphs of maximum degree two. Coy and Czumaj proved the following theorem:

▶ **Theorem 7** (Theorem 4.2 of [12]). *Let $G = (V, E)$ be an undirected simple graph with maximum degree $\Delta \leq 2$. One can deterministically find a matching $\mathcal{M}$ of G of size at least $m/8 = \Omega(m)$ in $O(1)$ MPC rounds with local space $\mathbf{S} = O(n^\delta)$, and global space $\mathbf{S}_{Global} = O(n)$.*

By extending their algorithm with the seed reduction technique mentioned earlier, we prove the following result:

▶ **Theorem 8.** *There exists an algorithm with the same properties as those in Theorem 7 using $\tilde{O}(n)$ total computation.*

We start by reviewing the main idea used in the algorithm proving Theorem 7.

Randomized Algorithm. The algorithm of Theorem 7 is based on derandomizing the following simple random process. Let $\{X_e : e \in E\}$ be a family of pairwise independent random variables with $X_e = 1$ with probability $p = 1/4$ and $X_e = 0$ otherwise. Now, let $\mathcal{M}$ be the matching that includes each edge e with $X_e = 1$ and $X_{e'} = 0$ for every neighboring edge e'. The expected size of this matching is:

$$\mathbb{E}[|\mathcal{M}|] = \sum_{e \in E} \Pr[e \in \mathcal{M}] \geq \sum_{e \in E} \Pr[X_e = 1] - \sum_{\substack{e' \in E \setminus \{e\}: \\ e' \cap e \neq \emptyset}} \Pr[X_e = 1 \cap X_{e'} = 1]$$
$$\geq m \cdot (p - 2p^2) \geq \frac{m}{8},$$

where the second inequality follows from pairwise independence of the random variable. Hence, they can be specified by a seed of length $2 \log n + O(1)$ by Lemma 5. As explained in [12], this allows to use the method of conditional expectation to deterministically find a matching of size at least $m/8$ in $O(1)$ MPC rounds.

Reducing the Seed Length. We next show how one can further reduce the seed length to $O(\log \log n)$. The main observation is that the above analysis holds as long as for any two neighboring edges the two corresponding variables are independent. This motivates the following approach. First, we assign to each edge e a color $c(e)$ from the set $\{1, 2, \ldots, C\}$ for $C = O(\log^2 n)$ by applying Lemma 6 such that two neighboring edges get assigned a different color. Let $\{X_c : c \in [C]\}$ be a family of pairwise independent random variables with $X_c = 1$ with probability $p = 1/4$ and $X_c = 0$ otherwise. We now include each edge e in $\mathcal{M}$ if $X_{c(e)} = 1$ and $X_{c(e')} = 0$ for every neighboring edge e'. The same calculations as above shows that $\mathbb{E}[\mathcal{M}] \geq \frac{m}{8}$.

MPC Algorithm. Now we are ready to present our deterministic MPC algorithm that proves Theorem 8. In the following, we say that something can be efficiently computed if there exists a deterministic MPC algorithm running in $O(1)$ rounds with local space $\mathbf{S} = O(n^\delta)$, global space $\mathbf{S}_{Global} = O(n)$ and using $\tilde{O}(n)$ total computation.

Let $\mathcal{H} = \{h : [C] \mapsto \{0, 1\}^2\}$ be a family of 2-wise independent hash functions of size at most $2^{2 \cdot \log C + O(1)} = \text{poly}(\log n)$ obtained using Lemma 5. Observe that each hash function $h \in \mathcal{H}$ defines a matching $\mathcal{M}(h)$ that includes each edge e with $h(c(e)) = 0$ and $h(c(e')) \neq 0$ for every neighboring edge e', where $h(i)$ denotes the length-2 bit sequence assigned to i by the corresponding integer in $\{0, \ldots, 3\}$.

The analysis of the randomized algorithm above implies that choosing a hash function h uniformly at random from $\mathcal{H}$ results in a matching of expected size at least $m/8$. In particular, this guarantees the existence of a hash function h^* with $\mathcal{M}(h^*) \geq m/8$. We efficiently compute $|\mathcal{M}(h)|$ for every $h \in \mathcal{H}$ and choose one *good* hash function that yields a matching of size at least $m/8$.

First, we efficiently compute the coloring c using Lemma 6. Next, we compute the approximate maximum matching in G by derandomizing the sampling approach analyzed above. Since the size of our family of pairwise independent hash functions is $\text{poly} \log n$, we can store one number per hash function on every machine. Each machine M_j, which is responsible for some edges $\mathcal{E}_j \subseteq [E]$, can compute locally the number of edges $\mathcal{M}^j(h) \subseteq E_j$ in the matching generated by $h \in \mathcal{H}$ within a single round. Then, we efficiently aggregate these numbers across all machines to compute the size of the matching $\mathcal{M}(h) = \sum_j \mathcal{M}^j(h)$ for every hash function h. The best $h^* \in \mathcal{H}$ for which $\mathcal{M}(h^*) \geq \frac{m}{8}$, breaking ties arbitrarily, yields our approximate maximum matching. Finally, let us note that the global memory occupied by the hash functions across all machines $\mathbf{M}$ is $\mathbf{M} \cdot |\mathcal{H}| \ll \mathbf{M} \cdot O(n^\delta) = O(n)$ and the overall computation performed to evaluate each hash function for every edge is $|\mathcal{H}| \cdot \text{poly}(\log n) \cdot O(n) = \tilde{O}(n)$.

4 Computation-Efficient Derandomization of Hitting Set

In this section, we give a deterministic MPC algorithm for the following hitting set variant defined in [12]:

▶ **Definition 9** (*Hitting Set for Leader Election*). *Let $S_1, \ldots, S_n$ be subsets of $[n]$ with $i \in S_i$ and $|S_i| = b$, for each $i \in [n]$. The goal is to find a (small) hitting set $\mathcal{L} \subseteq [n]$, that is, a set for which $S_i \cap \mathcal{L} \neq \emptyset$ holds for all $i \in [n]$.*

Coy and Czumaj [12] gave an algorithm with the same parameters as those of the random sampling approach in [7], except that they need large $\text{poly}(n)$ computation.

▶ **Theorem 10** (Theorem 5.6 of [12]). *Let b and n be integers with $\log^{10}(n) \le b \le n$. One can deterministically find a subset $\mathcal{L} \subseteq [n]$ that solves the Hitting Set for Leader Election problem with $|\mathcal{L}| \le O(n(\min\{b, \mathbf{S}\})^{-1/5})$ within a constant number of MPC rounds using local space $\mathbf{S} = O(n^\delta)$, global space $\mathbf{S}_{Global} = O(nb)$, and total computation $\text{poly}(n)$.*

We extend the randomized approach their algorithm relies on by using the method of alterations and reducing the amount of randomness needed to prove the following result:

▶ **Theorem 11.** *There exists an algorithm with the same properties as those in Theorem 10 with two differences. The total computation reduces to $O(n \cdot \text{poly}(b))$ and the global space increases to $O(n \cdot \text{poly}(b))$.*

We will show in Section 5 that the algorithm from Theorem 11 together with minor changes to the parameters of the connectivity algorithm results in a deterministic connectivity MPC algorithm with near-linear total computation.

Review of Hitting Set Algorithm of Coy and Czumaj

Consider adding each element to $\mathcal{L}$ with probability $p = b^{-1/5}$. Assuming full independence, the assumption $b \ge \log^{10}(n)$ together with a simple Chernoff Bound implies that $\mathcal{L}$ is a hitting set with high probability. The high probability bound still holds with $O(\log_b n)$-wise independence, but fails to hold with $o(\log_b n)$-wise independence. As n k-wise independent random variables require a seed length of $\Omega(k \log n)$, using $O(\log_b n)$-wise independence would not result in a seed length of $O(\log n)$, which is necessary for an $O(1)$ MPC round derandomization based on the method of conditional expectation. To shorten the seed length, the authors of [12] use so-called k-wise ε-approximately independent random variables for $k = 15 \log_b(n)$ and $\varepsilon = n^{-6}$. In particular, the starting point of their algorithm is the following theorem.

▶ **Theorem 12** ([12, Theorem 5.2]). *Let $\log^{10}(n) \le b \le n$, k be even with $k = 15 \log_b(n) \ge 4, \varepsilon = n^{-6}$, and $p = b^{-1/5}$. Then, if $X_1, X_2, \ldots, X_n$ are k-wise ε-approximately independent random variables with $X_i = 1$ with probability $b^{-1/5}$ and $X_i = 0$ otherwise. Then each of the following $n+1$ events hold with probability at least $1 - 9n^{-3}$:*

(1) $\sum_{j \in S_i} X_j > 0$ *for every $1 \le i \le n$, and*

(2) $\sum_{i=1}^{n} X_i \le 2nb^{-\frac{1}{5}}$.

Next, we explain our randomized approach, which bears some similarities with that of 12, and proceed to the reduction of its seed length and its deterministic implementation on an MPC with strongly sublinear memory.

Pairwise Analysis

As a first step, we show that a minor modification to their randomized hitting set algorithm results in a hitting set of expected size at most $2nb^{-1/5}$, assuming only pairwise independence. As before, each element joins $\mathcal{L}$ with probability $p = b^{-1/5}$. In expectation, $b \cdot p = b^{4/5}$ elements are sampled from each set. Using only pairwise independence and Chebyshev's inequality, this implies that a set is bad, i.e., no element is sampled from it, with probability at most $\frac{1}{b^{4/5}}$. This directly follows from the following lemma:

▶ **Lemma 13.** *Let $X_1, \ldots, X_n$ be pairwise independent random variables taking values in $[0, 1]$. Let $X = X_1 + \ldots + X_n$ and $\mu = \mathbb{E}[X]$. Then $\mathbb{V}\text{ar}[X] = \sum_{i=1}^{n} \mathbb{V}\text{ar}[X_i] \leq \mu$ and*

$$\Pr[|X - \mu| \geq \mu] \leq \frac{\sum_{i=1}^{n} \mathbb{V}\text{ar}[X_i]}{\mu^2} \leq \frac{1}{\mu}.$$

Hence, by adding for each unhit set an arbitrary element to $\mathcal{L}$, at most $n/b^{4/5}$ additional elements are added to $\mathcal{L}$ in expectation, resulting in a hitting set of expected size at most $n(b^{-1/5} + b^{-4/5})$.

Reducing The Seed Length

From the pairwise analysis above, we directly get a seed length of $O(\log n)$. Next, we show how to reduce the seed length to $O(\log b)$, which allows for a simple brute-force search. We again employ a coloring idea, which is based on the simple observation that we only require pairwise independence between elements contained in the same set. Hence, the goal is to color the elements with $\text{poly}(b)$ colors such that all elements in a given set S_i are colored with a different color.

In general, this may not be possible as there might exist elements which are contained in a lot of sets. Fortunately, a simple calculation shows that there exist at most n/b elements which are contained in more than b^2 different sets. Hence, by directly adding these elements to $\mathcal{L}$, we can assume "for free" that each element is contained in at most b^2 sets, which we will do from now on.

We can then obtain a coloring with the desired properties by finding a proper coloring in the graph $G_{conflict}$, defined as follows. The vertex set consists of one vertex for each of the n elements. Moreover, two elements are connected by an edge if there exists a set which contains both elements. Note that the maximum degree $\Delta_{conflict}$ of $G_{conflict}$ is upper bounded by b^3. This follows from our assumption that each element is contained in at most b^2 sets. Therefore, we can efficiently color $G_{conflict}$ with $C = O(\Delta_{conflict}^2 \log^2(n)) = O(b^6 \log^2 n)$ colors. For each $i \in [n]$, let $c(i)$ denote the color assigned to the i-th element. Note that it directly follows from the definition of $G_{conflict}$ that all elements in a given set are assigned a different color.

We are now ready to present our randomized process that produces a hitting set with the desired properties. Let $\{X_c : c \in [C]\}$ be a family of pairwise independent random variables with $X_c = 1$ with probability $p = b^{-1/5}$ and $X_c = 0$ otherwise. For simplicity, we assume that $1/p$ is a power of 2, i.e., there exists $\ell \in \mathbb{N}$ with $2^\ell = b^{1/5}$. According to Lemma 5, we can generate these random variables with a seed of length $2(\ell + \log C) + O(1) = O(\log b)$. Now, we add each element i with $X_{c(i)} = 1$ to $\mathcal{L}$. Then, for each set S_i with $\sum_{j \in S_i} X_{c(j)} = 0$, we add the element $i \in S_i$ to $\mathcal{L}$. By the analysis and discussion above, $\mathcal{L}$ is a hitting set of expected size $O(nb^{-1/5})$.

MPC Algorithm

It remains to discuss the MPC implementation, which will prove Theorem 11. In the following, we say that something can be efficiently computed if there exists a deterministic MPC algorithm running in $O(1)$ rounds with local space $= O(n^\delta)$, global space $O(n\text{poly}(b))$, and using $O(n\text{poly}(b))$ total computation.

In the preprocessing step, we add all elements which are contained in at least b^2 sets to the hitting set and remove all sets which contain at least one such element from consideration. The preprocessing step requires us to compute for each element in how many sets it is contained in. This can be done efficiently by using the colored summation primitive.

Next, we explain how to efficiently construct the graph $G_{conflict}$. We generate the edges of $G_{conflict}$ in two steps. First, each set $S = \{e_1, e_2, \ldots, e_b\}$ creates $\binom{b}{2}$ entries $\{\{e_i, e_j\} \colon i \neq j \in [b]\}$. This can easily be done with $\text{poly}(b)$ global space per set and $\min(\mathbf{S}, \text{poly}(b))$ local space in $O(1)$ rounds by using the primitives of Lemma 3. Hence, we can efficiently generate all these edges in parallel. Afterwards, we use the duplicate removal procedure of Lemma 3 to remove duplicate edges.

As $G_{conflict}$ has maximum degree b^3, we can use Lemma 6 to efficiently compute a coloring of $G_{conflict}$ with $C = O(b^6 \log^2 n) = \text{poly}(b)$ colors. As before, we denote with $c(i)$ the color assigned to the i-th element. For $\ell := \log_2(b^{1/5})$, let $\mathcal{H} = \{h \colon [C] \mapsto \{0,1\}^\ell\}$ be a family of 2-wise independent hash functions of size at most $2^{2(\ell + \log C) + O(1)} = \text{poly}(b)$ such that evaluating a function from $\mathcal{H}$ takes time $\text{poly}(\ell, \log C) = \text{poly}(\log b)$ time. Lemma 5 guarantees the existence of such a family.

For each function $h \in \mathcal{H}$, we define a hitting set $\mathcal{L}_h$ as follows. First, each element i with $h(c(i)) = 0$ is contained in $\mathcal{L}_h$, where $h(c(i))$ denotes the length-ℓ bit sequence for $c(i)$ by the corresponding integer in $\{0, \ldots, \ell - 1\}$. Moreover, if for a given set S_i no element contained in it was added in the first step, then we add element i to $\mathcal{L}_h$. The discussion above implies that there exists at least one hash function $h \in \mathcal{H}$ with $|\mathcal{L}_h| = O(nb^{-1/5})$. Using Lemma 3, it is easy to see that for a single hash function $h \in \mathcal{H}$, we can efficiently compute $\mathcal{L}_h$ and its size. As $\mathcal{H}$ only contains $\text{poly}(b)$ hash functions, this implies that we can efficiently compute $\mathcal{L}_h$ for every $h \in \mathcal{H}$. After we have done this, we can output the hitting set $\mathcal{L}_{h^*}$ of smallest size. As remarked above, $\mathcal{L}_{h^*}$ has size $O(nb^{-1/5})$, which finishes the proof.

5 Connectivity Algorithm

In this section, we discuss the necessary changes to the randomized connectivity algorithm of Behnezhad et al. [7] and its analysis in order to prove the main result of this paper.

The deterministic approximate matching from Section 3 is used to replace steps 5 and 6 of Algorithm 2 of [7]. The same modification was already done by [12] and they showed that the total number of vertices drop by a constant factor, assuming that no isolated vertex exists. Hence, by applying this modified algorithm $O(\log \log_{\frac{m}{n}} n)$ times, one can in $O(\log \log_{\frac{m}{n}} n)$ rounds ensure that $m \geq n \log^C n$, for a given constant C. All the steps of the modified deterministic algorithm can be implemented by invoking the primitives of Lemma 3 $O(1)$ times, which in particular ensures that the algorithm can be implemented with total computation $\tilde{O}(m)$. Hence, we can from now on assume that $m \geq n \log^C n$, for a given constant C. It remains to prove that Algorithm 1 of [7] can be implemented deterministically with the same asymptotic complexity and using $\tilde{O}(m)$ total computation, assuming $m \geq n \log^C(n)$ for a sufficiently large constant C. To this end, Coy and Czumaj proved the following lemma:

▶ **Lemma 14** ([12, Lemma 6.3]). *Let S_i denote the set of saturated vertices at level i after Step 2 of the* RelabelIntraLevel *routine in [7], let L_i denote the set of selected leaders at level i after Step 3 of the same execution of* RelabelIntraLevel*, let β_i denote the budget of vertices at level i, let $b(v)$ denote the budget of vertex v, and let γ, ε be arbitrary constants such that $0 < \gamma, \varepsilon < 1$. If we make the following modifications to* RelabelIntraLevel*:*

- *set $\beta_{i+1} := \beta_i \cdot (\min\{\beta_i, n^\varepsilon\})^{\gamma/4}$,*
- *replace Step 3 of* RelabelIntraLevel *with any MPC algorithm that in $O(1)$ rounds selects $O\left(\frac{|S_i|}{(\min\{\beta_i, n^\varepsilon\})^\gamma}\right)$ leaders for each level i with high probability or deterministically, and*
- *replace the budget update rule in Step 4 of* RelabelIntraLevel *with*

$$b(v) := b(v) \cdot (\min\{b(v), n^\varepsilon\})^{\gamma/4},$$

then the connectivity algorithm of [7] remains correct with the same asymptotic local and global space complexity.

We extend the above lemma to make it work with the deterministic hitting set from Section 4 by proving the following slight modification of it. The main technical challenge will be to ensure that our deterministic hitting set algorithm, which adds a polynomial factor (in b) increase in the memory and computation required, can still be run in parallel with linear global space and total computation.

▶ **Lemma 15.** *Let $c \geq 3$ be the smallest integer such that both the global space and the total computation required by the algorithm from Theorem 11 are bounded by $n \cdot b^c$, and let $\varepsilon = \delta/c$ so that $n^{c \cdot \varepsilon} \leq n^\delta$. The same result as that of Lemma 14 can be achieved with the following modifications to* RelabelIntraLevel*:*

- *set $\beta_{i+1} := \beta_i \cdot (\min\{\beta_i, n^\varepsilon\})^{\frac{\gamma}{4c}}$,*
- *replace Step 3 of* RelabelIntraLevel *with any MPC algorithm that in $O(1)$ rounds selects $O\left(\frac{|S_i|}{(\min\{\beta_i, n^\varepsilon\})^\gamma}\right)$ leaders for each level i with high probability or deterministically using at most $n\beta_i^c$ global space and total computation, and*
- *replace the budget update rule in Step 4 of* RelabelIntraLevel *with*

$$b(v) := b(v) \cdot (\min\{b(v), n^\varepsilon\})^{\frac{\gamma}{4c}},$$

and by replacing the initial budget $\left(\frac{m}{n}\right)^{1/2}$ assigned to each vertex with $\left(\frac{m}{n}\right)^{1/2c}$ in Algorithm 1 of [7]. Then, the connectivity algorithm of [7] remains correct with the same asymptotic local and global space complexity. Moreover, the resulting total computation is $O(m)$.

Proof. We need to show that all claims and lemmas involving the modified steps of Algorithm 1 of [7] do not affect its correctness nor its bounds on local and global memory. As in [12], we need to prove the following three key properties:

(1) for any vertex v, the value of $\ell(v)$ never exceeds $O(\log\log_{m/n} n)$ (cf. [7, Lemma 15]),
(2) the global space used is $O(\mathbf{S}_{Global})$ (cf. [7, Lemma 17]),
(3) the sum of the squares of the budgets does not exceed $O(\mathbf{S}_{Global})$ (cf. [7, Lemma 21]).

(1) Recall that the budget of each vertex is increased as $\beta_{i+1} := \beta_i \cdot (\min\{\beta_i, n^\varepsilon\})^{\frac{\gamma}{4c}}$ and that $\beta_0 = \left(\frac{m}{n}\right)^{1/2c}$. Since the budget of any vertex cannot exceed n, we have that there are at most $O(\log\log_{m/n} n)$ levels as required.

(2) Let n_i denote the number of vertices which *ever* reach level i over the course of the algorithm. In the proof of Lemma 17 [7], it is shown that the total sum of the budget increases over the course of the algorithm is $O(m)$, namely

$$\sum_{i=1}^{L} \beta_i n_i = O(m).$$

We extend this claim and prove that the total sum of the global space used by all hitting set instances over all iterations of the algorithm is bounded by $O(m)$, that is

$$\sum_{i=1}^{L} \beta_i^c \cdot n_i = O(m).$$

Analogously to [12], we first show that $\beta_{i+1}^c \cdot n_{i+1} \le \beta_i^c \cdot n_i$. We have that the number of vertices at level i removed from the graph (i.e., not marked as a leader) per vertex marked as leader is at least:

$$\frac{|S_i \setminus L_i|}{|L_i|} = \frac{|S_i| - |L_i|}{|L_i|} = \Omega\left((\min\{\beta_i, n^\varepsilon\})^\gamma\right) \gg (\min\{\beta_i, n^\varepsilon\})^{\gamma/2}.$$

It then follows that

$$\begin{aligned} \beta_{i+1}^c \cdot n_{i+1} &= \left(\beta_i \cdot (\min\{\beta_i, n^\varepsilon\})^{\frac{\gamma}{4c}}\right)^c n_{i+1} \\ &< \left(\beta_i^c \cdot (\min\{\beta_i, n^\varepsilon\})^{\frac{\gamma}{4}}\right)\left(n_i(\min\{\beta_i, n^\varepsilon\})^{-\gamma/2}\right) \\ &\le \beta_i^c \cdot n_i. \end{aligned}$$

Using the fact that the maximum possible level for a vertex is $L = O(\log\log n)$, we obtain

$$\sum_{i=1}^{L} \beta_i^c \cdot n_i \le L \cdot (\beta_0^c \cdot n_0) \le O(\log\log n) \cdot \left(\frac{m}{n}\right)^{\frac{1}{2}} \cdot n,$$

where the last inequality comes from the fact that $\beta_0 = \left(\frac{m}{n}\right)^{\frac{1}{2c}}$. Note that we can assume that $m \ge n\log^{20c}(n)$ and therefore each vertex has an initial budget of $\beta_0 = (m/n)^{1/2c} \ge \log^{10}(n) \gg O(\log\log n)$, as required by Theorem 11. This yields

$$O(\log\log n) \cdot \left(\frac{m}{n}\right)^{\frac{1}{2}} \cdot n \ll \left(\frac{m}{n}\right)^{\frac{1}{2}} \cdot \left(\frac{m}{n}\right)^{\frac{1}{2}} \cdot n = O(m).$$

(3) Follows by the same line of reasoning as in property (2).

By the choice of c, repeating the same calculations as in property (b) proves that the total computation required by running our deterministic hitting set algorithm over all instances in each iteration of the algorithm does not exceed $O(m)$. Moreover, Lemma 3 implies that all the other steps of the algorithm can be implemented with total computation $\tilde{O}(m)$. ◀

We are now ready to prove our main result.

Proof of Theorem 2. We apply Lemma 15 using our Hitting Set for Leader Election algorithm from Theorem 11 setting $\gamma = \frac{1}{5}$ (Note that $m \ge n\log^C(n)$ for a sufficiently large constant C implies $\beta_0 \ge \log^{10}(n)$). Then, it follows directly from Lemma 6.4 of [12] combined with Lemma 15 that copies of our hitting set algorithms can be run in parallel, for each possible level and in a constant number of rounds within optimal global space and $\tilde{O}(m)$

total computation. Thus, we proved that all relevant aspects of the proof of correctness have been adjusted in comparison to [12, 7]. Finally, as noted in [12], our extension of Lemma 15 in [7] proves that the number of iterations remains asymptotically the same and that the deterministic algorithms replacing the $O(1)$-round random sampling approach take asymptotically the same number of rounds. Thus, we conclude that the round complexity is not affected. ◀

References

1. Noga Alon, László Babai, and Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of Algorithms*, 7(4):567–583, 1986. doi:10.1016/0196-6774(86)90019-2.
2. Noga Alon and Joel H Spencer. *The probabilistic method.* John Wiley & Sons, 2016.
3. Alexandr Andoni, Zhao Song, Clifford Stein, Zhengyu Wang, and Peilin Zhong. Parallel graph connectivity in log diameter rounds. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 674–685, 2018. doi:10.1109/FOCS.2018.00070.
4. Sepehr Assadi, Xiaorui Sun, and Omri Weinstein. Massively parallel algorithms for finding well-connected components in sparse graphs. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 461–470, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293611.3331596.
5. Philipp Bamberger, Fabian Kuhn, and Yannic Maus. Efficient deterministic distributed coloring with small bandwidth. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, pages 243–252, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3382734.3404504.
6. Soheil Behnezhad, Sebastian Brandt, Mahsa Derakhshan, Manuela Fischer, MohammadTaghi Hajiaghayi, Richard M. Karp, and Jara Uitto. Massively parallel computation of matching and mis in sparse graphs. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 481–490, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293611.3331609.
7. Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, and Vahab Mirrokni. Near-optimal massively parallel graph connectivity. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1615–1636, 2019. doi:10.1109/FOCS.2019.00095.
8. J.Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979. doi:10.1016/0022-0000(79)90044-8.
9. Keren Censor-Hillel, Merav Parter, and Gregory Schwartzman. Derandomizing local distributed algorithms under bandwidth restrictions. *Distributed Computing*, 33(3):349–366, June 2020. doi:10.1007/s00446-020-00376-1.
10. Moses Charikar, Weiyun Ma, and Li-Yang Tan. Brief announcement: A randomness-efficient massively parallel algorithm for connectivity. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 431–433, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3465084.3467951.
11. Benny Chor and Oded Goldreich. On the power of two-point based sampling. *Journal of Complexity*, 5(1):96–106, 1989. doi:10.1016/0885-064X(89)90015-0.
12. Sam Coy and Artur Czumaj. Deterministic massively parallel connectivity. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2022, pages 162–175, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519935.3520055.
13. Artur Czumaj, Peter Davies, and Merav Parter. Simple, deterministic, constant-round coloring in the congested clique. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, pages 309–318, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3382734.3405751.

14 Artur Czumaj, Peter Davies, and Merav Parter. Component stability in low-space massively parallel computation. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 481–491, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3465084.3467903.

15 Artur Czumaj, Peter Davies, and Merav Parter. Graph sparsification for derandomizing massively parallel computation with low space. *ACM Trans. Algorithms*, 17(2), May 2021. doi:10.1145/3451992.

16 Artur Czumaj, Peter Davies, and Merav Parter. Improved deterministic (delta+1) coloring in low-space mpc. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 469–479, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3465084.3467937.

17 Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. doi:10.1145/1327452.1327492.

18 Guy Even, Oded Goldreich, Michael Luby, Noam Nisan, and Boban Veličković. Efficient approximation of product distributions. *Random Structures & Algorithms*, 13(1):1–16, 1998. doi:10.1002/(SICI)1098-2418(199808)13:1<1::AID-RSA1>3.0.CO;2-W.

19 Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, Cliff Stein, and Zoya Svitkina. On distributing symmetric streaming computations. *ACM Trans. Algorithms*, 6(4), September 2010. doi:10.1145/1824777.1824786.

20 Mohsen Ghaffari, Christoph Grunau, and Ce Jin. Improved MPC Algorithms for MIS, Matching, and Coloring on Trees and Beyond. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 34:1–34:18, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DISC.2020.34.

21 Mohsen Ghaffari and Fabian Kuhn. Derandomizing Distributed Algorithms with Small Messages: Spanners and Dominating Set. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing (DISC 2018)*, volume 121 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:17, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DISC.2018.29.

22 Mohsen Ghaffari, Fabian Kuhn, and Jara Uitto. Conditional hardness results for massively parallel computation from distributed lower bounds. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1650–1663, 2019. doi:10.1109/FOCS.2019.00097.

23 Mohsen Ghaffari and Jara Uitto. Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. In *Proceedings of the 2019 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1636–1653, 2019. doi:10.1137/1.9781611975482.99.

24 Michael T. Goodrich. Communication-efficient parallel sorting. *SIAM Journal on Computing*, 29(2):416–432, 1999. doi:10.1137/S0097539795294141.

25 Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In Takao Asano, Shin-ichi Nakano, Yoshio Okamoto, and Osamu Watanabe, editors, *Algorithms and Computation*, pages 374–383, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-25591-5_39.

26 Tomasz Jurdziński and Krzysztof Nowicki. MST in O(1) Rounds of Congested Clique. In *Proceedings of the 2018 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2620–2632, 2018. doi:10.1137/1.9781611975031.167.

27 Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the 2010 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 938–948, 2010. doi:10.1137/1.9781611973075.76.

28 Fabian Kuhn. Weak graph colorings: Distributed algorithms and applications. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 138–144, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1583991.1584032.

29 Jakub Lacki, Vahab S. Mirrokni, and Michal Wlodarczyk. Connected components at scale via local contractions. *CoRR*, abs/1807.10727, 2018. arXiv:1807.10727.

30 Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: A method for solving graph problems in mapreduce. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 85–94, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1989493.1989505.

31 Christoph Lenzen and Roger Wattenhofer. Brief announcement: Exponential speed-up of local algorithms using non-local communication. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 295–296, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1835698.1835772.

32 Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992. doi:10.1137/0221015.

33 Sixue Cliff Liu, Robert E. Tarjan, and Peilin Zhong. *Connected Components on a PRAM in Log Diameter Time*, pages 359–369. Association for Computing Machinery, New York, NY, USA, 2020. doi:10.1145/3350755.3400249.

34 Michael Luby. Removing randomness in parallel computation without a processor penalty. *Journal of Computer and System Sciences*, 47(2):250–286, 1993. doi:10.1016/0022-0000(93)90033-S.

35 Michael Luby and Avi Wigderson. Pairwise independence and derandomization. *Foundations and Trends in Theoretical Computer Science*, 1(4):237–301, 2006. doi:10.1561/0400000009.

36 Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms.* Cambridge university press, 1995.

37 Danupon Nanongkai and Michele Scquizzato. Equivalence classes and conditional hardness in massively parallel computations. *Distributed Computing*, 35(2):165–183, 2022. doi:10.1007/s00446-021-00418-2.

38 Krzysztof Nowicki. A deterministic algorithm for the mst problem in constant rounds of congested clique. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 1154–1165, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3406325.3451136.

39 Merav Parter and Eylon Yogev. Congested Clique Algorithms for Graph Spanners. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing (DISC 2018)*, volume 121 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 40:1–40:18, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DISC.2018.40.

40 Prabhakar Raghavan. Probabilistic construction of deterministic algorithms: Approximating packing integer programs. *Journal of Computer and System Sciences*, 37(2):130–143, 1988. doi:10.1016/0022-0000(88)90003-7.

41 Tim Roughgarden, Sergei Vassilvitskii, and Joshua R. Wang. Shuffles and circuits (on lower bounds for modern parallel computation). *J. ACM*, 65(6), November 2018. doi:10.1145/3232536.

42 Mark N. Wegman and J. Lawrence Carter. New classes and applications of hash functions. In *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, pages 175–182, 1979. doi:10.1109/SFCS.1979.26.

Fault Tolerant Coloring of the Asynchronous Cycle

Pierre Fraigniaud
Université Paris Cité, CNRS, IRIF, F-75013, Paris, France

Patrick Lambein-Monette[1]
Université Paris Cité, CNRS, IRIF, F-75013, Paris, France

Mikaël Rabie
Université Paris Cité, CNRS, IRIF, F-75013, Paris, France

Abstract

We present a wait-free algorithm for proper coloring the n nodes of the asynchronous cycle C_n, where each crash-prone node starts with its (unique) identifier as input. The algorithm is independent of $n \geqslant 3$, and runs in $\mathrm{O}(\log^* n)$ rounds in C_n. This round-complexity is optimal thanks to a known matching lower bound, which applies even to synchronous (failure-free) executions. The range of colors used by our algorithm, namely $\{0, \ldots, 4\}$, is optimal too, thanks to a known lower bound on the minimum number of names for which renaming is solvable wait-free in shared-memory systems, whenever n is a power of a prime. Indeed, our model coincides with the shared-memory model whenever $n = 3$, and the minimum number of names for which renaming is possible in 3-process shared-memory systems is 5.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Mathematics of computing → Graph coloring; Computer systems organization → Dependable and fault-tolerant systems and networks; Theory of computation → Models of computation

Keywords and phrases graph coloring, LOCAL model, shared-memory model, immediate snapshot, renaming, wait-free algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.23

Related Version *Full Version*: https://arxiv.org/abs/2207.11198

Funding *Pierre Fraigniaud*: additional support from the project ANR-20-CE48-0006 (DUCAT).

1 Introduction

1.1 Motivation

Two forms of coloring tasks are at the core of distributed computing. One is *vertex-coloring* [8] in the framework of synchronous distributed network computing [29]. The other is *renaming* [3] in the framework of asynchronous shared-memory distributed computing [7]. For both tasks, each process starts with its own *identifier* as input, which is supposed to be unique in the system, and must compute a *color* as output. The identifiers are supposed to be in a large range of values (typically of size $\mathrm{poly}(n)$), while the colors should lie in a restricted range of values, typically $\{0, \ldots, k-1\}$ for some $k \geqslant 1$. Depending on the context, k may be an absolute constant, or may depend on parameters of the system, like the maximum degree Δ of the network, or even the total number n of processes. In the context of network computing, the outputs must properly color the underlying graph of the network, i.e., any two neighboring nodes must output distinct colors. In the context of shared-memory computing, each process must output a color that is unique in the system, i.e., different from the color of any other process.

[1] Correponding author

© Pierre Fraigniaud, Patrick Lambein-Monette, and Mikaël Rabie;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 23; pp. 23:1–23:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

On the negative side, it is "hard" to color cycles of even size using only two colors in a distributed manner [26], in the sense that $\Omega(n)$ synchronous rounds of communication are required to solve this problem in the n-node cycle C_n (n even; 2-coloring an odd length cycle is impossible). A synchronous *round* consists of (1) an exchange of information between the two end-points of every edge in the network, and (2) a local computation at every node. Similarly, renaming the n processes of an asynchronous shared-memory system in a wait-free manner using a palette with fewer than $2n-1$ names (i.e., k-renaming with $k < 2n-1$) is impossible [6, 14, 24] whenever n is a power of a prime number ($n = 6$ is the smallest integer for which this bound does not hold [15]). *Wait-free* essentially means that each process terminates in a bounded number of write/read steps, independently of the asynchronous scheduling of the $n-1$ other processes, i.e., independently of the interleaving of read and write operations in the shared memory.

On the positive side, it is known that 3-coloring the n-node cycle C_n for $n \geqslant 3$ can be achieved in $\frac{1}{2}\log^* n + \mathrm{O}(1)$ synchronous rounds thanks to *deterministic coin tossing*, an efficient color-reduction technique due to Cole and Vishkin [17][2]. This bound is tight, as no algorithms can 3-color the n-node cycle in less than $\frac{1}{2}\log^* n - 1$ rounds, thanks to Linial's celebrated lower bound [26]. In shared-memory systems, while $(2n-2)$-renaming is impossible wait-free for infinitely many values of n, $(2n-1)$-renaming can be achieved wait-free for all values of $n \geqslant 2$ [3].

The above results are at the core of two separate lines of intensive research. One line studies extensions of 3-coloring the synchronous cycle, in particular $(\Delta+1)$-coloring arbitrary networks of maximum degree Δ; see, e.g., [9, 20, 23, 30] for recent contributions in this field. This line also studies variants of $(\Delta+1)$-coloring, including, for example, Δ-coloring, edge-coloring, weak-coloring, defective coloring; see, e.g., [8, 21, 22, 28]. The other line of research studies variants of renaming (e.g., long-lived [1, 5]), renaming in different shared-memory or message-passing models (e.g., [2, 16]), and the search for algorithms using fewer names whenever n is not a power of a prime [4, 15].

1.2 Objective

Our aim is to study coloring tasks in a framework relaxing two strong assumptions made in the aforementioned contexts. First, it relaxes the "all-to-all assumption" of the shared-memory model, which enables some form of global communication between the processing nodes, or processes. Second, our framework relaxes the "synchrony assumption" of the LOCAL [29] model of network computing, where the processes proceed in lock-step, in the sense that we allow *processes* to be fully asynchronous and crash-prone, while we keep reliable and instantaneous *communications* (the latter is in contrast with the classic asynchronous model known as *message-passing* [19], where, in addition, the delivery of messages is itself asynchronous). Specifically, we consider a round-based, asynchronous computing model in the n-node cycle C_n, where each *round* of a process consists of the following sequence of operations: (1) writing in its local register, (2) reading the local registers of its two neighbors in C_n, and (3) updating its local state.

The difference with the standard LOCAL model, in which vertex-coloring is typically studied, is that the rounds are asynchronous. That is, the scheduler may allow some processes to perform many rounds while other processes may perform just a few rounds, or even no

[2] For every $x > 0$, let $\log^{(0)} x := x$ and, for $k \geqslant 0$ such that $\log^{(k)} x > 0$, let $\log^{(k+1)} x := \log_2(\log^{(k)} x)$; $\log^* x$ is then defined as the smallest $k \geqslant 0$ such that $\log^{(k)} x \leqslant 1$.

rounds. Moreover, the operations performed during a round are also asynchronous, e.g., a process can write, read, and then spend a lot of time idle before changing state. In particular, the cycle may become disconnected, and processes may become isolated, due to processes that are very slow, or even crashed (a crash is a full-stop form of failure: a crashed process stops functioning, and does not recover). As a consequence, information may propagate poorly in the network due to slow or crashed processes.

The difference between our setting and typical models (e.g., shared memory) used for studying renaming [7] is that the processes do not share a single array of single-writer/multiple-reader registers. Instead, only processes sitting on adjacent nodes in C_n can read each-other's registers. Thus, instead of having processes perform snapshot operations – i.e., read the registers of all processes at once – or even immediate snapshots – i.e., write a value *and* read everything all at once – each process is restricted to *local* (immediate) snapshots, i.e., that only read the registers of its neighbors in the cycle.

We seek to address a few basic questions about this model. Is wait-free proper vertex-coloring at all possible in C_n? That is, can the n processes of the asynchronous cycle pick colors distinct from those of their neighbors in a bounded number of computational steps? If yes, what is the smallest range of colors that make it possible to color the asynchronous cycle C_n? And what is the smallest number of asynchronous rounds that a process may have to perform in order to achieve this task?

Note that it is a priori unclear whether wait-free proper vertex-coloring is at all possible in the asynchronous cycle, even if allowing a large range of colors (but less than the number n of processes). Indeed, there are very similar problems which are not solvable in this framework. An example is *maximal independent set* (MIS). MIS and 3-coloring are reducible one to another in the cycle under the synchronous failure-free setting [26]; in contrast, MIS is *not* solvable wait-free in the asynchronous crash-prone version of the LOCAL model considered in this paper (Property 1). Indeed, as we detail further down, a wait-free algorithm for MIS could be simulated in the asynchronous shared-memory model for solving *strong symmetry-breaking* wait-free, which was proved impossible in [6].

1.3 Our Results

We describe a wait-free algorithm for proper coloring the n processes of the asynchronous crash-prone cycle C_n. So, wait-free proper vertex-coloring is possible in C_n, as opposed to, e.g., MIS. Our algorithm is independent of $n \geqslant 3$, and each process performs $O(\log^* n)$ asynchronous rounds in C_n. The round complexity of our algorithm is therefore asymptotically optimal, thanks to Linial's lower bound [26], which holds for the executions of our model that are synchronous and failure-free.

The range of colors used by our algorithm, namely $\{0, \dots, 4\}$, is optimal too for the class of all cycles, thanks to the aforementioned minimum number $2n - 1$ of names for which renaming is solvable wait-free in shared-memory systems, whenever n is a power of a prime. Indeed, in the specific case of the cycle C_3, our model coincides with the shared-memory model with $n = 3$ processes, which implies that proper coloring C_3 with less than five colors is impossible.

To our knowledge, our algorithm is the first distributed coloring algorithm designed for a framework combining the following two sources of difficulties: on the one hand, the possibility of crash failures in a fully asynchronous setting, and, on the other hand, a network limiting direct communications between processes.

Our Technique. Our main algorithm, given in Algorithm 3, has two components.

The first component of Algorithm 3 is introduced standalone in Algorithm 2. It bears some resemblance to the rank-based $(2n-1)$-renaming algorithm (see [7, Algorithm 55], and [3, Step 4 in Algorithm A]). It is a wait-free 5-coloring algorithm for C_n, i.e., in each of its executions over a cycle of length $n \geqslant 3$, the processes that perform enough computational steps output a color in the set $\{0, \ldots, 4\}$, and no two neighboring processes output the same color. However, Algorithm 2 is slow, in the sense that its running time may be as large as the longest sub-path of the cycle along which process identifiers are increasing, which can be as large as $\Theta(n)$.

The second component of Algorithm 3 uses and modifies the identifiers, in parallel to the first component. This quickly shortens such increasing sub-paths, until their length less than some constant $L \leqslant 10$, in a manner directly inspired from Cole and Vishkin's method [17]. Each process starts with its input identifiers, and successively tries to adopt new ones taken from increasingly smaller ranges of identifiers, by performing $O(\log^* n)$ identifier-reductions. As this reduction process goes on, the identifiers might not remain unique in the cycle, but we ensure that they nonetheless maintain a proper coloring, i.e., adjacent processes always hold distinct identifiers. This invariant is difficult to enforce in an asynchronous environment, and we resort to a synchronization mechanism by which a process awaits a "green light" from both of its neighbors each time it seeks to change its identifier.

The second component of our algorithm is thus not wait-free by itself, since processes are constantly waiting for "green lights" from their neighbors. However, it offers *starvation free* progress [25]: termination is guaranteed whenever all processes perform infinitely many computational steps. Our core result is that the interaction between the two components, i.e., between the (wait-free) first component and the (starvation-free) second component, remains itself wait-free, and has a running time $O(\log^* n)$.

We can decompose the description of the first component further, into a starvation-free subcomponent that looks for a color a_p for every process p, which does not collide with the colors of the neighbors of p with greater identifiers, and in another subcomponent that looks for a (potentially different) color b_p for process p, which doesn't collide with the colors of *any of* p's neighbors. The latter subcomponent offers *obstruction-free* progress [25]: termination is guaranteed whenever processes are scheduled to take multiple consecutive steps alone. As obstruction-free progress and starvation-free progress are both strictly weaker than wait-free progress, it is of independent interest that we are able to bootstrap a wait-free algorithm from subcomponents that aren't themselves wait-free.

1.4 Related Work

The closest recent contributions related to the current work are [13], and the follow-up work [18], which consider a related model, albeit a distinct one. The former provides a distributed algorithm for 3-coloring the ring, while the latter provides a distributed algorithm for $(\Delta+1)$-coloring graphs with maximum degree Δ. The two papers assume n asynchronous crash-prone processes occupying the n nodes of a reliable *and synchronous* network. That is, the communications remain synchronous, and a message emitted by a node u at round r reaches all nodes at distance d from u at round $r+d$. Moreover, no messages are lost, in the sense that a late-waking process will find all messages that passed through the node it occupies. Because it "decouples" the *computing layer* from the *communication layer*, this model is called DECOUPLED in [13].

The DECOUPLED model is stronger than the fully asynchronous model considered in this paper. In fact, [18] shows that, for every task (e.g., vertex-coloring, edge-coloring, maximal independent set, etc.), if there exists an algorithm for solving that task in the

LOCAL model in $t = \mathrm{O}(\mathrm{polylog}\ n)$ rounds, then there exists an algorithm for solving the task in the DECOUPLED model in $\mathrm{O}(t)$-round. In contrast, some tasks that are trivial in the LOCAL and the DECOUPLED model become impossible in our fully asynchronous model, like 3-coloring C_3 (Property 3), or computing a maximal independent set (Property 1).

The model considered in this paper bears similarities with some models used in the context of *self-stabilization*. Many papers (see, e.g., [9, 10, 11, 12]) have addressed the design of self-stabilizing algorithms for 3-coloring the cycles, or for $(\Delta+1)$-coloring graphs with maximum degree Δ. Self-stabilization assumes that the processes start in an arbitrarily bad state (all variables can be corrupted). The objective is to design algorithms which, starting from an arbitrary initial configuration, eventually compute a legal configuration (e.g., a configuration in which the colors assigned to the nodes form a proper coloring) whenever no failures occur during a sufficiently long period. In contrast, we assume an initial configuration in which variables are correctly set. However, we do not assume that the system will be failure-free during the execution of the algorithm, and the presence of crash-failures should not prevent the correct processing nodes from computing a solution. While 3-coloring the cycle C_n is possible in a self-stabilizing manner for all $n \geqslant 3$, k-coloring C_3 is impossible in our fully asynchronous model for $k < 5$ (Property 3).

2 Model and Observations

In this section, we first describe an asynchronous variant of the (synchronous) LOCAL model, which we will call the *partial immediate snapshot* model for reasons that will soon become apparent. The model can be viewed as a sort of asynchronous message-passing on a graph with a local broadcast communication primitive and instantaneous message delivery. Equivalently, it can be viewed as a shared-memory system where access to the shared memory is mediated by a graph; we adopt the latter approach in our description. We define what is a *round* in this model, what is the *round complexity* of an algorithm, what it means to be *wait-free*, and then we provide lower bounds on the round-complexity and on the range of colors for the problem of wait-free vertex-coloring the cycle.

2.1 Operational Model

The model is described for the cycle, but it can directly be extended to any network. Specifically, we consider asynchronous wait-free computing in the n-node cycle C_n, where the processes attached to each node exchange information between neighbors using single-writer/multiple-reader registers. Each process is a deterministic (infinite) state machine. All n processes are initially asleep; they may wake up at any time, and not all processes need to wake up, or to take enough steps to terminate correctly (i.e., processes are prone to fail-stop faults). Awakened processes proceed asynchronously, each with the objective of computing a color in $\{0, \ldots, 4\}$. We focus on wait-free tasks, i.e., where a process that takes enough steps is guaranteed to terminate, regardless of the scheduling of the other processes, so as to prevent deadlocks resulting from a process waiting for an event which will never occur because another process has crashed.

Just like for the standard coloring and renaming tasks, the only input given to a process p is its identifier X_p, which is an integer in the range $[0, \mathrm{poly}(n)]$ that is unique in the system. We do not assume that the processes are aware of the length n of the cycle, nor even of an upper bound on n. Every process proceeds with a sequence of exchanges of information with its neighbors until some condition is satisfied by its local state, at which point it terminates and outputs a color obtained by applying some function to this local state.

Immediate snapshots. Let us first recall how communication works using a standard *immediate snapshot* communication primitive. In this model, the n processes $p_1, \ldots, p_n$ communicate through n single-writer/multiple-readers registers $R_1, \ldots, R_n$, initialized with an initial value $\perp$. Every process can read all registers, but each process p_i is the single writer in register R_i, $i \in \{1, \ldots, n\}$. Each process p_i goes through a (possibly infinite) sequence of write-read-update steps, where in each step it: (1) writes a value in register R_i, (2) reads the content of all registers, and (3) performs a private computation. Taken together, these three steps constitute an asynchronous *round* of process p_i.

Each of the rounds is instantaneous, but the time elapsed between two of p_i's rounds may be arbitrarily long. For example, process p_i may perform many rounds while p_j performs none, in which case p_i will read the same value in register R_j every time, possibly $\perp$ if p_j hasn't awakened yet. Conversely, in-between two consecutive rounds of p_i's, there may be faster processes that performed many writes in their registers.

The value read by a process in a register R_j is the one written by p_j in its most recent round. Multiple processes may perform a round at the same time. In this case, the system behaves as if each of these processes first wrote a value in its own register, then all processes read all registers, and, finally, they all performed their private computation. Note that distinct processes may be at distinct rounds of their execution. For example, one process may be just starting, i.e., in its first round, while another may already have been running for some time, and so be at a later round.

Local immediate shapshots. Our model simply adds a graph to the above, which mediates which registers a process is able to read. For example, in the cycle, a process only reads three registers: its own register, and the register of each of its two neighbors. We do not assume a coherent notion of left and right, i.e., each node assigns an arbitrary order to the registers of its neighbors.

In this paper, we do not assume that the registers are bounded. Nevertheless, our algorithms only manipulate a constant number of variables using $\mathrm{O}(\log n)$ bits each.

2.2 Schedules and complexity

In our model, an execution is entirely characterized by the code of each process, the graph (here, the cycle C_n), the input identifiers of each process, as well as the activation patterns of each process. The latter is captured by the collection of n increasing sequences $t_p^{(1)}, t_p^{(2)}, \ldots$ of positive integers, one for each process $p \in [n]$, where $t_p^{(i)}$ denotes the time in which process p performs its i-th round.

As multiple processes may be performing rounds simultaneously, let us introduce, for $t \geqslant 1$, the set $\sigma(t)$ of activated processes at time t. We set: $p \in \sigma(t) \iff \exists i \geqslant 1 : t_p^{(i)} = t$. The *schedule* of an execution is the infinite sequence $\sigma = \sigma(1), \sigma(2), \ldots$ An execution of a given algorithm on the cycle C_n is thus determined by the schedule σ and the input identifiers $(X_p)_{p\in[n]}$.

We will say that a process $p \in \sigma(t)$ is *working* if the stopping condition of p has not been fulfilled before time t. This leads us to define, for any schedule σ, the restricted schedule $\overline{\sigma}$ of working processes:

$$\overline{\sigma}(t) \coloneqq \{\, p \in \sigma(t) \mid p \text{ has not fulfilled the stopping condition at time} \leqslant t-1 \,\}.$$

An execution *terminates* if there exists some time t^* such that $\overline{\sigma}(t) = \varnothing$ for all subsequent times $t \geqslant t^*$, i.e., if eventually all processes stop working. Note that a process stops working according to two possible scenarios: it may have been activated sufficiently many times for

allowing it to fulfill the stopping condition, or it was not activated after some time t, before it fulfilled the stopping condition. The latter scenario models the crash of a process (at time t, or earlier in the execution). The *round complexity* of a terminating execution is then defined as

$$\max\{\, i \in \mathbb{N} \mid \exists p \in [n] : p \in \overline{\sigma}(t_p^{(i)}) \,\}.$$

The running time of an algorithm over the cycle C_n is then the supremum of the round complexity *for all possible executions*, i.e., all possible identifier assignments and schedules. Informally, the running time corresponds to the maximal number of times a process can be activated before it is guaranteed to terminate. An algorithm is then *wait-free* if its running time is finite.

2.3 Lower Bounds and Impossibility Results

We complete this section by a couple of observations on the round complexity, and on the range of colors used by wait-free vertex-coloring algorithms for the cycle. Before that, we formalize the fact that, as claimed in the introduction, the maximal independent set (MIS) problem cannot be solved in the asynchronous cycle. Solving the MIS problem requires that, at the end of every execution, (1) every node that terminates and outputs 0 is neighbor of at least one node that terminates and output 1, and (2) no two neighboring nodes that terminate output 1.

▶ **Property 1.** *For every $n \geqslant 3$, MIS in the n-node cycle C_n, cannot be solved wait-free in our model.*

Proof. The proof is by reduction from the *strong symmetry-breaking* (SSB) problem, which cannot be solved wait-free in the asynchronous shared-memory model (see [6, Theorem 11]). We show that if there were an algorithm solving MIS in the n-node cycle, then there would exist an algorithm for SSB in the n-node shared-memory system. Recall that SSB requires that (1) if all processes terminate, then at least one processes outputs 0, and at least one process outputs 1, and (2) in every execution, at least one process outputs 1. By way of contradiction, let $\mathcal{A}$ be an algorithm solving MIS in C_n. The n processes of shared-memory system can simulate the algorithm $\mathcal{A}$ as follows. Process p_i, $i = 0, \ldots, n-1$, simulates the execution of the algorithm $\mathcal{A}$ at the node of C_n with identifier i, and with neighbors the nodes with identifiers $i \pm 1 \bmod n$, which are simulated by processes $p_{i\pm 1 \bmod n}$, respectively. Since the algorithm $\mathcal{A}$ solves MIS, it guarantees that, if all processes terminate, then at least one outputs 0, and at least one outputs 1. Moreover, in every execution of the algorithm $\mathcal{A}$, a node that terminates and is isolated (none of its neighbors terminated) must output 1, and a node that terminates and has a neighbor that terminates is such that either itself outputs 1, or at least one of its neighbors outputs 1. This guarantees that, in every execution, at least one process output 1. The two conditions for solving SSB are therefore fulfilled by simulating the algorithm $\mathcal{A}$, and thus $\mathcal{A}$ cannot exist. ◀

We now show that the round-complexity of our vertex-coloring algorithm is optimal.

▶ **Property 2.** *For every $k \geqslant 2$, the round-complexity of any wait-free algorithm for k-coloring the vertices of the n-node cycles C_n, $n \geqslant 3$, requires $\Omega(\log^* n)$ rounds in the state model.*

Proof. This directly follows from [26], which proved that, in synchronous and failure-free executions, i.e., $\sigma(t) = \{\, 1, \ldots, n \,\}$ for all $t \geqslant 1$, k-coloring the vertices of the n-node cycles C_n, requires $\Omega(\log^* n)$ rounds. ◀

Finally, we show that the range of colors used by our algorithm is optimal.

▶ **Property 3.** *If a wait-free algorithm k-colors all asynchronous cycles $\mathcal{C} = \{ C_n \mid n \geqslant 3 \}$, then $k \geqslant 5$.*

Proof. The partial shared-memory model in the cycle coincides with the standard shared-memory model when $n = 3$, since the cycle C_3 is complete. The result thus directly follows from the impossibility for $n = 3$ asynchronous processes to solve renaming wait-free using fewer than five names in an immediate snapshot shared-memory model [6, 14]. ◀

Note that Property 3 leaves open the possibility that, for specific values of n, fewer colors could be used to color the cycle C_n wait-free, the same way the lower bound $2n - 1$ on the number of names for renaming only holds when n is a power of a prime. However, a generic algorithm capable of proper coloring every cycle C_n, for all $n \geqslant 3$, must use at least 5 colors, as our algorithm does. Nevertheless, the shared-memory model with immediate snapshots does not coincide with our model when $n > 3$, and thus it may well be the case that fewer than 5 colors could be used for some specific values of $n > 3$, although we conjecture that this is not the case.

3 Asynchronously coloring the cycle in linear time

Here we develop asynchronous coloring algorithms, and show that **a)** they guarantee wait-free progress – i.e., a process will terminate in all executions, provided that it is activated sufficiently many times – and **b)** they are correct – i.e., the graph induced by the terminating processes is properly colored by the output colors of these processes. These algorithms have a poor runtime complexity of $\mathrm{O}(n)$ steps when compared to state-of-the-art algorithms in the LOCAL model, which terminate in $\mathrm{O}(\log^* n)$ synchronous rounds. We will achieve a similar runtime complexity in the next section by augmenting our wait-free algorithms with a mechanism that speeds up termination.

We first present an algorithm that uses a 6-color palette. Although it uses one extra color when compared to the theoretical minimum of 5 colors required to color the cycle C_3, this allows us to illustrate some of our main algorithmic ingredients. We then present another wait-free algorithm that colors any cycle using a 5-color palette. Some of the longer proofs of this section can be found in Appendix B.

3.1 Warm-up: using a palette of 6 colors

In Algorithm 1, we present a simple algorithm for wait-free coloring any cycle C_n ($n \geqslant 3$), using the six colors in the set $\{ (a, b) \in \mathbb{N} \times \mathbb{N} \mid a + b \leqslant 2 \}$. Given a process p, we denote by X_p its identifier, and by q and q' its two neighboring process in the cycle. We denote by $u \sim v$ the fact that processes u and v are neighbors in C_n. A process p, with neighbors q and q', is said to be *locally extremal* (with respect to the identifiers) if either $X_p > \max\{ X_q, X_{q'} \}$ or $X_p < \min\{ X_q, X_{q'} \}$.

Intuitively, Algorithm 1 guarantees that locally extremal processes quickly terminate, by sticking to one of the two components a_p or b_p of their color $c_p = (a_p, b_p)$ (Lemma 7). Termination then propagates throughout the cycle, due to the wait-free nature of the algorithm (Lemmas 6 and 7). Given an initial coloring of C_n provided by the nodes' identifiers, we will show that the worst-case convergence time of a process is determined by its distance to its nearest local extrema, which is bounded by $\mathrm{O}(\min\{ n, \max_p X_p - \min_q X_q \})$, which yields a linear convergence time.

Algorithm 1 6-coloring algorithm, code for process p with neighbors q and q'.

```
1 Input: X_p ∈ ℕ
2 Initially:
3    c_p = (a_p, b_p) ← (0,0) ∈ ℕ × ℕ
4 Forever:
5    write(X_p, c_p) and read((X_q, c_q), (X_q', c_q'))  ▷ local immediate snapshot
6    if c_p ∉ { c_q, c_q' } then return(c_p)
7    else
8        a_p ← min ℕ ∖ { a_u | (u ∼ p) ∧ (X_u > X_p) }
9        b_p ← min ℕ ∖ { b_u | (u ∼ p) ∧ (X_u < X_p) }
```

▶ **Theorem 4.** *In any execution of Algorithm 1 over the cycle C_n with a proper coloring provided by the values $(X_p)_{p\in[n]}$ given to the processes as input, we have:*
Termination: *every process terminates after having been activated at most $\lfloor 3n/2 \rfloor + 4$ times;*
6-color palette: *every process that terminates outputs a color in the set $\{ (a,b) \mid a+b \leqslant 2 \}$;*
Correctness: *the outputs properly color the graph induced by the terminating processes in C_n.*

The rest of the subsection is dedicated to the proof of Theorem 4. Recall that, in a schedule σ, a process $p \in \sigma(t)$ is *working* in t if it has not returned before t. Once a working process returns, it no longer partakes in the execution.

Notation. We will adopt the following notation for all algorithms throughout the paper. If x_p is a variable used by process p, we use $x_p(t)$ to denote the value of x_p *in p's memory*, at the *end* of time t, and we use $\widehat{x}_p(t)$ to denote the value of x_p *visible to p's neighbors* at the end of time t. Let $x_p(0)$ be given by the initialization of the algorithm, and let $\widehat{x}_p(0) = \bot$. By definition, we have

$$\widehat{x}_p(t) = \begin{cases} x_p(t-1) & p \in \overline{\sigma}(t) \\ \widehat{x}_p(t-1) & p \notin \overline{\sigma}(t) \end{cases} \tag{1}$$

▶ **Lemma 5.** *Let $t \geqslant 0$, and let $p \in \overline{\sigma}(t)$. We have $c_p(t) \notin \{ \widehat{c}_q(t) \mid q \sim p \}$, and process p returns at time t if and only if $c_p(t) = c_p(t-1)$.*

Proof. Process p does not update c_p when it returns, and so $c_p(t) = c_p(t-1)$ whenever p returns at time t. Let us then assume that $p \in \overline{\sigma}(t)$ does *not* return at time t, and let q be one of p's neighbors. If q has not yet been activated then $\widehat{c}_q(t) = \bot \neq \widehat{c}_p(t)$. If q has been already activated then, since the inputs form an initial proper coloring, we either have $X_p > X_q$ or $X_p < X_q$. In the former case, we have $a_p(t) \neq \widehat{a}_q(t)$, and in the latter case, we have $b_p(t) \neq \widehat{b}_q(t)$. Either way, we have $c_p(t) \neq \widehat{c}_q(t)$, and so $c_p(t) \neq c_p(t-1)$, since $c_p(t-1) = \widehat{c}_p(t) \in \{ \widehat{c}_q(t), \widehat{c}_{q'}(t) \}$ ◀

Lemma 5 provides us with an effective characterization of $\overline{\sigma}$: for every $t \geqslant 0$ and every $p \in [n]$,

$$p \in \overline{\sigma}(t) \iff \forall t' < t : \big(p \in \sigma(t') \implies c_p(t') \neq c_p(t'-1)\big). \tag{2}$$

The next lemma formalize the intuition that a process terminates fast, unless the execution is "very interleaved".

▶ **Lemma 6.** *Let p be a process that is working at times t_1 and $t_2 > t_1$, but is not activated at any time $t \in [t_1 + 1, t_2]$. If neither of p's neighbors is working in the time interval (t_1, t_2), then process p returns at time t_2.*

Proof. The result directly follows from Lemma 5, using the fact that $c_p(t_1) \notin \{ \widehat{c}_q(t_1) \mid q \sim p \}$ and $\widehat{c}_p(t_2) = c_p(t_1)$. ◀

As the next lemma shows, a process cannot be prevented from returning by only one of its neighbors.

▶ **Lemma 7.** *Let process p be activated at times $t_1 < t_2 < t_3 < t_4$, but not at any other time $t \in (t_1, t_4)$. If $a_p(t_1) = a_p(t_2) = a_p(t_3) = a_p(t_4)$, and X_p is not a local minimum, then p returns at time at most t_4. The same holds if $b_p(t_1) = b_p(t_2) = b_p(t_3) = b_p(t_4)$ and X_p is not a local maximum.*

Note that, even though $X_p(t)$ remains constant throughout the execution, the public value $\widehat{X}_p(t)$ doesn't, as initially its value is $\perp$. To analyze executions of Algorithm 1, let us introduce the sets

$$N_p^+(t) \coloneqq \{ q \sim p \mid \widehat{X}_q(t) > \widehat{X}_p(t) \} \text{ and } N_p^-(t) \coloneqq \{ q \sim p \mid \widehat{X}_q(t) < \widehat{X}_p(t) \}.$$

We furthermore define the sets

$$A_p(t) \coloneqq \begin{cases} \bigcup_{q \in N_p^+(t)} \left(\widehat{A}_q(t) \cup \{ \widehat{X}_q(t) \} \right) & p \in \overline{\sigma}(t) \\ A_p(t-1) & p \notin \overline{\sigma}(t) \end{cases} \tag{3}$$

and

$$B_p(t) \coloneqq \begin{cases} \bigcup_{q \in N_p^-(t)} \left(\widehat{B}_q(t) \cup \{ \widehat{X}_q(t) \} \right) & p \in \overline{\sigma}(t) \\ B_p(t-1) & p \notin \overline{\sigma}(t) \end{cases} \tag{4}$$

where $A_p(0) = B_p(0) = \varnothing$, and where the sets $\widehat{A}_p(t), \widehat{B}_p(t)$ are defined according to Equation (1). The set $A_p(t)$ contains all processes that p has heard of at time t, and that are linked to p through a subpath of C_n where process identifiers are increasing. Symmetrically, the set $B_p(t)$ contains processes that p has heard of, and that are linked to p through a subpath where identifiers are decreasing.

▶ **Lemma 8.** *Let $t \in \mathbb{N}$, and let $p \in [n]$ be a process. For every $x \in A_p(t)$, we have $\widehat{X}_p(t) < x$, and, for every $x \in B_p(t)$, we have $\widehat{X}_p(t) > x$.*

▶ Remark 9. This will be used in the next section, where we present a procedure for speeding up Algorithm 2 by reducing the space of colors initially provided to the nodes thanks to their identifiers. On the other hand, the claim $\widehat{X}_p(t) > \max B_p(t)$ doesn't generalize under the same weaker condition.

In the case where X_p does not change, we can notice that $A_p(t)$ and $B_p(t)$ are increasing, inclusion-wise, with time. Moreover, the elements of $A_p(t)$ correspond to increasing identifiers X_q following a path from p (decreasing in the case of $B_p(t)$). Hence, $|A_p(t)|$ has a size bounded by the length of the longest path of increasing identifiers from p.

If a process $p \in \overline{\sigma}(t)$ fails to return in time t, the sets $A_p(t)$ and $B_p(t)$ help us compute its next color $c_p(t)$.

▶ **Lemma 10.** *For any time $t \geqslant 1$, if a process $p \in \overline{\sigma}(t)$ fails to return at time t, then:*

1. *if $\left|N_p^+(t)\right| \leqslant 1$, then $a_p(t) \equiv |A_p(t)| \bmod 2$;*
2. *if $\left|N_p^-(t)\right| \leqslant 1$, then $b_p(t) \equiv |B_p(t)| \bmod 2$.*

As a direct consequence of Lemma 10, we get the following.

▶ **Lemma 11.** *Let $t \geqslant 0$, and let $p \in [n]$ be non-extremal a process. If $p \in \overline{\sigma}(t)$, but p fails to return at time t, then we have $A_p(t) \neq A_p(t-1)$ or $B_p(t) \neq B_p(t-1)$.*

Proof. Using Lemma 10, if $A_p(t) = A_p(t-1)$ and $B_p(t) = B_p(t-1)$ then $c_p(t) = c_p(t-1)$, and so by Lemma 5 process p returns, a contradiction. ◀

This leads us to the following complexity bound for processes that are not local extrema. It relies on the distance of a process to its closest local extrema along monotone paths. Let q_i, $i = 0, \ldots, k+1$, be a set of distinct processes, excepted possibly $q_{k+1} = q_0$. Let us assume that these processes form a subpath of C_n, or possibly the entire cycle C_n if $q_{k+1} = q_0$. That is, $q_0 \sim q_1 \sim q_2 \cdots \sim q_k \sim q_{k+1}$. Let us assume that $X_{q_0} < X_{q_1}$ and $X_{q_k} < X_{q_{k+1}}$, but $X_{q_1} > X_{q_2} > \cdots > X_{q_k}$, i.e., process q_1 is locally maximal, process q_k is locally maximal, and for $i \in \{1, \ldots, k\}$, process q_i is at monotone distance $i-1$ from its closest local maximum q_1, and at monotone distance $k-i$ from its closest local minimum q_k.

▶ **Lemma 12.** *Let $p \in [n]$ be a non-extremal process, and let ℓ and ℓ' be the monotone distances from p to its closest extremal processes. Process p returns after at most $\min\{3\ell, 3\ell', \ell+\ell'\}+4$ activations.*

Proof. We know from Remark 9 that $A_p(t)$ is increasing with time, and that its size is bounded by ℓ. Thanks to Lemma 10, we have that $a_p(t)$ is determined by the size of $A_p(t)$. It follows that $a_p(t)$ changes at most $\ell + 1$ times. Symmetrically, $b_p(t)$ changes at most ℓ' times. By Lemma 7, we get that a process p cannot be activated more than 3 times while keeping the same value for $a_p(t)$. It follows that process p can be activated at most $3\ell + 4$ times before it returns. Symmetrically, p can be activated at most $3\ell' + 4$ times before it returns. Finally, from Lemma 11, we get that p can be activated at most $\ell + \ell' + 1$ times before it returns. ◀

This last results allows us to conclude.

Proof of Theorem 4. As a direct corollary of Lemma 7, that local extrema return after at most 4 steps: a maximum will maintain $a(t) = 0$, and a minimum, $b(t) = 0$. For the other nodes, Lemma 12 gives us the complexity, knowing that $\min\{\ell, \ell'\}$ is bounded by $\lfloor 3n/2 \rfloor$. ◀

▶ Remark 13. Lemma 12 states that the complexity of Algorithm 1 is linear in the length of the longest chain of processes $p_1 \sim p_2 \sim \cdots$ that is monotone for the identifiers, i.e., $X_{p_1} > X_{p_2} > \cdots$. Throughout this section, we have assumed that the processes start with their identifiers as input, and that each identifier is unique in the network, i.e., $X_p \neq X_q$ whenever $p \neq q$. Note however that Theorem 4 only requires that identifiers form a *proper coloring*, i.e., $X_p \neq X_q$ whenever $p \sim q$. In this case, the length of a monotone chain is bounded by the number of initial colors, and so is the convergence of Algorithm 1. In the Section 4, we exploit this property to dramatically accelerate our algorithms by dynamically adjusting the “identifiers” X_p themselves, using a modification of Cole and Vishkin's classic algorithm [17], initially designed for the PRAM model, but easily adapted to the LOCAL model. As we shall see, its adaptation to the asynchronous setting is more subtle.

3.2 Saving one color: wait-free 5-coloring the cycle

Here we present, in Algorithm 2, another wait-free coloring algorithm for the cycle, which only uses a palette of five colors. As already noted, when the graph is a clique, asynchronous coloring is identical to the renaming problem using an immediate snapshot communication primitive, which implies that asynchronously coloring the cycle C_3 *requires* at least a five-colors palette. Our algorithm is thus optimal in terms of colors for the class $\mathcal{C} = \{ C_n \mid n \geqslant 3 \}$ of all cycles.

Algorithm 2 5-coloring algorithm, code for process p with neighbors q and q'.

```
1  Input: X_p ∈ ℕ
2  Initially:
3  |  a_p, b_p ← 0 ∈ ℕ
4  Forever:
5  |  write(X_p, a_p, b_p) and read((X_q, a_q, b_q), (X_q', a_q', b_q'))  ▷ local imm. snap.
6  |  P+ ← { u ∈ { q, q' } | X_u > X_p }
7  |  C+ ← { a_u | u ∈ P+ } ∪ { b_u | u ∈ P+ }
8  |  C ← { a_q, b_q, a_q', b_q' }
9  |  if a_p ∉ C then return(a_p)
10 |  else if b_p ∉ C then return(b_p)
11 |  else
12 |  |  a_p ← min ℕ ∖ C+
13 |  |  b_p ← min ℕ ∖ C
```

▶ **Theorem 14.** *In any execution of Algorithm 2 over the cycle C_n with a proper coloring provided by the values $(X_p)_{p\in[n]}$ given to the processes as input, we have:*
Termination: *every process terminates after having been activated at most $\mathrm{O}(n)$ times;*
5-color palette: *every process that terminates outputs a color in the set $\{ 0, \ldots, 4 \}$;*
Correctness: *the outputs properly color the graph induced by the terminating processes in C_n.*

From the algorithm, we immediately deduce the following characterization of when a process returns a value.

▶ **Lemma 15.** *Let $t \geqslant 1$, and let $p \in \overline{\sigma}(t)$ be a process with neighbors q and q'. Let $C := \{ \widehat{a}_q(t), \widehat{b}_q(t), \widehat{a}_{q'}(t), \widehat{b}_{q'}(t) \}$. We have $b_p(t) \notin C$, and process p returns at time t if and only if $a_p(t-1) \notin C$ or $b_p(t-1) \notin C$.*

Note that, as a consequence of the previous lemma, $b_p(t) \neq b_p(t-1)$ unless $p \in \overline{\sigma}(t)$ returns at time t, and so Lemma 6 continues to hold for Algorithm 2.

Defining the sets $A_p(t)$ as we did for Algorithm 1, we get the following sufficient condition for a process to terminate.

▶ **Lemma 16.** *Suppose that process $p \in [n]$ is* not *a local minimum for the identifiers. If p is activated at times $t_1 < t_2 < t_3 < t_4$, and $A_p(t_1) = A_p(t_2) = A_p(t_3) = A_p(t_4)$, then p returns at time at most t_4.*

▶ **Lemma 17.** *Let $p \in [n]$ be a process that is* not *a local minimum for the identifiers, and let ℓ denote the monotone distance from p to the closest maximal process. Process p returns after at most $3\,\ell + 4$ activations.*

Proof. This is a direct consequence of the previous lemma: for p to keep working, its set $A_p(t)$ must increase at least every 4 activations. The claim follows. ◀

Proof of Theorem 14. Thanks to Lemma 17, processes that are *not* local minima return after a number of steps that is at most $\lfloor 3n/2 \rfloor + 4$. Local minima terminate at most one step after their two neighbors have terminated, i.e., in at most $3n+8$ rounds. The proper coloring is an immediate consequence of Lemma 15. ◀

4 From Linear Time to Almost Constant Time

Here, we augment Algorithm 2 with a mechanism designed to reduce X_p, initially set to the identifier of the process. As the identifiers[3] will now be evolving through time, we will say that a process p, with neighbors q, q', is a local extremum at time $t \geqslant 1$ if $\widehat{X}_p(t) > \widehat{X}_q(t), \widehat{X}_{q'}(t)$. The resulting algorithm, displayed as Algorithm 3, 5-colors the cycle C_n in $\mathrm{O}(\log^* n)$ steps. Some of the longer proofs of this section can be found in Appendix B.

The intuition for Algorithm 3 is as follows. Every process p essentially runs Algorithm 2 unchanged, and stops whenever this algorithm terminates. However, in parallel, every process p updates its identifier X_p, initially equal to the identifier of p, *à la* Cole and Vishkin using a reduction function f defined hereafter. This helps to shorten long monotone chains of identifiers down to a constant length, speeding up the convergence of Algorithm 2. This addition to the algorithm is blocking, as, to maintain a proper coloring of the identifiers X_p (which is crucial for the wait-free coloration algorithm), every process p must wait for the approval of both its neighbors each time p wants to update its identifier, through the use of a local counter r_p which tracks the number of times process p tried to pick a smaller identifier. If all processes advance "almost synchronously", then they quickly (in $\mathrm{O}(\log^* n)$ steps) reach a stage where the remaining monotone chains of identifiers are all shorter than a constant $L \leqslant 10$. From then on, the algorithm behaves as Algorithm 2, and all processes terminate in $\mathrm{O}(L)$ steps, that is, in constant time. The crux of the proof is therefore to show that slow processes cannot delay the convergence of fast processes too much. Indeed, a slow process may delay other processes, but if it blocks them during too many iterations (with respect to the reduction of the identifiers X_p), then the system starts behaving as Algorithm 2, and neighboring processes actually quickly terminate. On the other hand, if a process is only "moderately slow", and allows its neighbors to make some progress on the reduction of their identifiers X_p, then other processes use this property for breaking symmetry, and they stop waiting for the slow process.

4.1 Reducing identifiers with deterministic coin-tossing

The considerable speedup achieved in comparison to Algorithm 2 relies on an identifier-reduction function $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, adapted from Cole and Vishkin's algorithm [17], defined as follows. For any natural number Z, we denote its binary decomposition by $Z = \sum_{k \in \mathbb{N}} Z_k 2^k$, and its length by $|Z| \coloneqq \lceil \log_2(Z+1) \rceil$. Given two natural numbers X and Y, we then set

$$f(X,Y) = 2i + X_i \quad \text{where } i \coloneqq \min\{\, |X|, |Y| \,\} \cup \{\, k \in \mathbb{N} \mid X_k \neq Y_k \,\} \tag{5}$$

As $f(x,y) \leqslant 2|x| + 1 = \mathrm{O}(\log(x))$, one reaches a constant fixed point after $\mathrm{O}(\log^* n)$ iterate calls to f, which gives the following. Recall that, for $k \in \mathbb{N}$, the k-th iterate of a function $F : A \to A$ is recursively defined as $F^{(0)}(x) = x$ and, for $k \geqslant 1$, $F^{(k)} = F \circ F^{(k-1)}$.

[3] For simplicity, we continue to refer to $X_p(t)$ as process p's "identifier", even though it is now possible that $X_q(t) = X_p(t)$ for some other process $q \nsim p$.

▶ **Lemma 18.** *Let $F : [1, +\infty) \to [1, +\infty)$ be the function $x \mapsto F(x) = 2\lceil \log(x+1) \rceil + 1$. There exists $\alpha > 0$ such that, for every $x \geqslant 1$, there exists $t \leqslant \alpha \log^* x$ such that $F^{(t)}(x) < 10$.*

▶ **Lemma 19.** *Let $x, y \in \mathbb{N}$. If $x > y \geqslant 10$, then $f(x, y) < y$.*

Proof. Let $\ell = |y|$. By assumption, we have $\ell \geqslant 4$. If $\ell = 4$, then $f(x, y) \leqslant 2\ell + 1 = 9 < y$. If $\ell \geqslant 5$, then we have $y \geqslant 2^{\ell-1}$, and so $y - f(x, y) \geqslant 2^{\ell-1} - 2\ell - 1 > 0$, where the last inequality is because $2^z > 4z + 2$ whenever $z \geqslant 5$. ◀

The proper coloring maintained by the function f relies on the following Cole and Vishkin-like property.

▶ **Lemma 20.** *Let $x, y, z \in \mathbb{N}$. If $x > y > z$, then $f(x, y) \neq f(y, z)$.*

Proof. Let $f(x, y) = 2i^* + x_{i^*}$. For all $i < i^*$, $x_i = y_i$, and if $i^* < |y|$ then $x_i \neq y_i$. Suppose that $f(y, z) = f(x, y)$. Then $y_{i^*} = x_{i^*}$, and by the above $i^* \geqslant |y| \geqslant |z|$. In this case, $y_i = z_i$ for all $i < |y|$, and thus $y = z$, contradicting our assumption $y > z$. ◀

4.2 5-coloring the cycle in near-constant time

Algorithm 3 Fast 5-coloring algorithm, code for process p with neighbors q and q'.

```
1  Input: X_p ∈ ℕ
2  Initially:
3  └ a_p, b_p, r_p ← 0 ∈ ℕ
4  Forever:
5  │ write(X_p, r_p, a_p, b_p) and read((X_q, r_q, a_q, b_q), (X_q', r_q', a_q', b_q'))
6  │ if a_p ∉ { a_q, b_q, a_q', b_q' } then return(a_p)
7  │ else if b_p ∉ { a_q, b_q, a_q', b_q' } then return(b_p)
8  │ else
9  │ │ a_p ← min ℕ ∖ { a_u, b_u | (u ∼ p) ∧ (X_u > X_p) }
10 │ │ b_p ← min ℕ ∖ { a_q, b_q, a_q', b_q' }
11 │ │ if (r_p < ∞) ∧ (r_p ⩽ min{ r_q, r_q' }) then
12 │ │ │ if min{ X_q, X_q' } < X_p < max{ X_q, X_q' } then
13 │ │ │ │ r_p ← r_p + 1
14 │ │ │ │ Y ← f(X_p, min{ X_q, X_q' })          ▷ f given in Equation (5)
15 │ │ │ │ if Y < min{ X_q, X_q' } then X_p ← Y
16 │ │ │ else
17 │ │ │ │ r_p ← ∞
18 │ │ │ │ if X_p < min{ X_q, X_q' } then
19 │ │ │ │ └ X_p ← min{ X_p, min(ℕ ∖ { f(X_q, X_p), f(X_q', X_p) }) }
```

▶ **Theorem 21.** *In any execution of Algorithm 3 over the cycle C_n with a proper coloring provided by the values $(X_p)_{p \in [n]}$ given to the processes as input:*

Termination: *every process terminates after having been activated at most $O(\log^* n)$ times;*

5-color palette: *every process that terminates outputs a color in the set $\{0, \ldots, 4\}$;*

Correctness: *the outputs properly color the graph induced by the terminating processes in C_n.*

A crucial ingredient in the proof of correctness is to establish that the coloring provided by the evolving values of the local variables X_p, $p \in [n]$, is always proper throughout any execution.

▶ **Lemma 22.** *Let $p, q \in [n]$ be neighboring processes. For every $t \in \mathbb{N}$, if $\widehat{X}_p(t) \neq \perp$ then $\widehat{X}_p(t) \neq \widehat{X}_q(t)$.*

When discussing executions of Algorithm 3, we say that a process p is *blocked* at time t if p has not yet returned at time t and $r_p(t) = \widehat{r}_p(t) < \infty$. Since the value of X_p changes only if r_p increases, we have $X_p(t) = \widehat{X}_p(t)$ whenever process p is blocked at time t. A process p that *is not* blocked at time t, will write a new value for $\widehat{r}_p(t)$ at its next activation. Moreover, p writes a new value for $\widehat{X}_p(t)$ as well, unless p satisfies specific properties: X_p is a local maximum, X_p is a local minimum, or p has a neighbor q with $\widehat{X}_q < 10$. Note that, before its first activation, every process p is unblocked, as $r_p(0) = 0 \neq \widehat{r}_p(0) = \perp$.

Every process that takes sufficiently many non-blocked steps, namely $\Omega(\log^* n)$ steps, quickly reduces its identifier X_p until either X_p, or the identifier X_q of one of its neighbors q becomes smaller than 10. At this stage of the execution, monotone chains of identifiers will cease to evolve after an additional constant number of steps. Once the monotone chains of identifiers cease to evolve, the analysis developed in the previous section shows that processes terminate in a number of steps that is not larger than the length of monotone chains of identifiers, which is itself bounded by a constant $L \leqslant 10$. In other words, when all processes take $\Omega(\log^* n)$ non-blocked steps, they terminate in an additional $\mathrm{O}(1)$ steps.

In the following, we then focus on the case where the identifiers of the processes are still greater than 10, and we will show fast convergence is guaranteed even in the presence of blocked processes. Indeed, the main difficulty in proving Theorem 21 is to deal with blocked processes. Mainly, we show that a process quickly terminates whenever it is not blocked at *too many* steps.

▶ **Lemma 23.** *Let $p \in [n]$ be a process. For all $t \geqslant 1$, if $p \in \overline{\sigma}(t)$ and $\widehat{X}_p(t)$ is a local maximum in some time t, then $\widehat{X}_p(t')$ is a local maximum for all $t' \geqslant t$.*

Proof. Local maxima never update their identifiers; other processes might, but only to decrease them. The claim follows. ◀

▶ **Lemma 24.** *Let $p \in [n]$ be a process, and let q, q' be its two neighboring processes in the cycle. Let us assume that processes p and q are blocked at some time t_0, with $\widehat{r}_q(t_0) < \widehat{r}_p(t_0)$, $\widehat{X}_q(t_0) > \widehat{X}_p(t_0) > \widehat{X}_{q'}(t_0)$, and $\widehat{X}_p(t_0) \geqslant 10$. Additionally, let us assume that process q remains blocked throughout the whole time interval $[t_0, t_1)$ for some $t_1 > t_0$, but becomes activated and unblocked at time t_1. Then, one of the following claims holds:*

- *X_q is a local maximum at time t_1.*
- *If process q is activated again at some time $t_2 > t_1$, then X_p will become a local maximum as soon as p is activated at a time $t \geqslant t_2$.*

▶ **Lemma 25.** *Let $k \geqslant 1$, and let $q_0 \sim q_1 \sim \cdots \sim q_k$ be a sequence of $k+1$ distinct processes in the cycle. Let $t_0 \in \mathbb{N}$, and let $t_1 \in \mathbb{N} \cup \{\infty\}$ with $t_1 > t_0$. Let us assume that (1) for every $t \in [t_0, t_1)$, $q_0 \notin \overline{\sigma}(t)$, (2) processes $q_1, \ldots, q_k$ are blocked at time t_0, with $\widehat{r}_{q_0}(t_0) < \widehat{r}_{q_1}(t_0) < \cdots < \widehat{r}_{q_k}(t_0) < \infty$, and (3) $\widehat{X}_{q_k}(t_0) \geqslant 10$. Then, for every $i \in \{1, \ldots, k\}$, process q_i terminates after having been activated at most $3i + 1$ times in the time interval $[t_0, t_1)$.*

▶ **Lemma 26.** *Let $p, q \in [n]$ be two neighboring processes. If X_q is a local maximum at time $t_0 \in \mathbb{N}$, and if $\widehat{r}_q(t_0) = \infty$, then p terminates after having been activated $\mathrm{O}(\log^* n)$ times during the time interval $[t_0, \infty)$.*

▶ **Lemma 27.** *Let $p \in [n]$ be a process. If p is blocked at every time $t \in [t_0, t_1)$, and if p takes 4 steps during that interval, then p takes up to $\mathrm{O}(\log^* n)$ steps in $[t_0, \infty)$ before terminating.*

We are now ready to show Theorem 21.

Proof of Theorem 21. For a process p, there are two possible paths, both leading to p returning:

1. Process p never gets blocked. By Lemma 18, if a process updates its identifier up to $\mathrm{O}(\log^* n)$ times, its identifier ends up in the interval $[0, 10]$. Therefore, after $\mathrm{O}(\log^* n)$ rounds, either X_p becomes a local maximum, or $X_p \leqslant 10$. In the first case, it stays a maximum by Lemma 23, its $a_p(t)$ remains constant, and p terminates after 4 rounds, thanks to Lemma 16. In the second case, it will stay at distance at most 10 from a local minimum. As the processes of this path will no longer change their X_{-}, Lemma 17 allows us to conclude.
2. Process p becomes blocked. This can happen after at most $\mathrm{O}(\log^* n)$ rounds (otherwise we would end up in the previous case). Lemma 27 ensures that at most $\mathrm{O}(\log^* n)$ rounds will happen before p returns.

This complete the proof that 5-coloring the asynchronous cycles C_n, $n \geqslant 3$, can be achieved in $O(\log^* n)$ rounds. ◀

5 Conclusion and future works

We have presented a wait-free distributed algorithm for proper coloring the n nodes of the asynchronous cycle C_n, for every $n \geqslant 3$. This algorithm performs in $\mathrm{O}(\log^* n)$ rounds, which is optimal, thanks to Linial's lower bound [26] that applies to the synchronous execution. The algorithm uses 5 colors to proper color any cycle C_n, $n \geqslant 3$, matches the minimum number 5 of colors required to properly color the asynchronous cycle C_3 [6, 14, 24]. Even if, for $n > 3$, the existence of a 3-coloring algorithm is not directly ruled out by [6, 14, 24], we conjecture that k-coloring the n-node cycle C_n requires $k \geqslant 5$ for every $n \geqslant 3$.

A natural extension of this work is to consider wait-free coloring arbitrary graphs. Note that, by the renaming lower bound, coloring graphs with maximum degree Δ requires a palette of at least $2\Delta + 1$ colors whenever $\Delta + 1$ is a power of a prime. This is because the shared memory model and the model in this paper coincide in the case of coloring the clique of $n = \Delta + 1$ nodes. We do not know if $2\Delta + 1$ colors suffice for properly coloring all graphs of maximum degree Δ in a wait-free manner. It is however easy to extend Algorithm 1 to graphs with maximum degree Δ, for every $\Delta \geqslant 2$, using a range of colors of size $O(\Delta^2)$ (see Appendix A). The running time of this algorithm may however be as large as the one of Algorithm 1, i.e., $O(n)$ rounds. In the synchronous setting, there is a $O(\Delta^2)$-coloring algorithm performing in $O(\log^* n)$ rounds [26] in any graph. However, the techniques used in the synchronous setting for reducing the number of colors from $O(\Delta^2)$ to $\Delta + 1$ (see [27]) seem hard to transfer to the asynchronous setting.

More generally, it would be interesting to characterize which classical graph problems studied in synchronous failure-free networks admit wait-free solutions for asynchronous networks, and which do not. And, for those solvable wait-free, what are their round-complexities? For instance, 5-coloring can be solved wait-free in the asynchronous cycle, in $O(\log^* n)$ rounds, while maximal independent set (MIS) cannot be solved at all in asynchronous cycles.

References

1. Yehuda Afek, Hagit Attiya, Arie Fouren, Gideon Stupp, and Dan Touitou. Long-lived renaming made adaptive. In *18th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 91–103, 1999. doi:10.1145/301308.301335.
2. Dan Alistarh, Hagit Attiya, Rachid Guerraoui, and Corentin Travers. Early deciding synchronous renaming in $o(\log f)$ rounds or less. In *19th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, LNCS 7355, pages 195–206. Springer, 2012. doi:10.1007/978-3-642-31104-8_17.
3. Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, July 1990. doi:10.1145/79147.79158.
4. Hagit Attiya, Armando Castañeda, Maurice Herlihy, and Ami Paz. Bounds on the step and namespace complexity of renaming. *SIAM Journal on Computing*, 48(1):1–32, 2019. doi:10.1137/16M1081439.
5. Hagit Attiya and Arie Fouren. Polynominal and adaptive long-lived ($2k$-1)-renaming. In *14th International Conference on Distributed Computing (DISC)*, LNCS 1914, pages 149–163. Springer, 2000. doi:10.1007/3-540-40026-5_10.
6. Hagit Attiya and Ami Paz. Counting-based impossibility proofs for set agreement and renaming. *Journal of Parallel and Distributed Computing*, 87:1–12, 2016. doi:10.1016/j.jpdc.2015.09.002.
7. Hagit Attiya and Jennifer Welch. *Distributed Computing*. Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, Inc., Hoboken, NJ, USA, April 2004. doi:10.1002/0471478210.
8. Leonid Barenboim and Michael Elkin. *Distributed Graph Coloring: Fundamentals and Recent Developments*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2013. doi:10.2200/S00520ED1V01Y201307DCT011.
9. Leonid Barenboim, Michael Elkin, and Uri Goldenberg. Locally-iterative distributed $(\delta + 1)$-coloring below szegedy-vishwanathan barrier, and applications to self-stabilization and to restricted-bandwidth models. In *37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 437–446, 2018. doi:10.1145/3212734.3212769.
10. Samuel Bernard, Stéphane Devismes, Maria Gradinariu Potop-Butucaru, and Sébastien Tixeuil. Optimal deterministic self-stabilizing vertex coloring in unidirectional anonymous networks. In *23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–8, 2009. doi:10.1109/IPDPS.2009.5161053.
11. Jean R. S. Blair and Fredrik Manne. An efficient self-stabilizing distance-2 coloring algorithm. *Theoretical Computer Science*, 444:28–39, 2012. doi:10.1016/j.tcs.2012.01.034.
12. Lélia Blin, Laurent Feuilloley, and Gabriel Le Bouder. Brief Announcement: Memory Lower Bounds for Self-Stabilization. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 37:1–37:3, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.DISC.2019.37.
13. Armando Castañeda, Carole Delporte-Gallet, Hugues Fauconnier, Sergio Rajsbaum, and Michel Raynal. Making local algorithms wait-free: the case of ring coloring. *Theory of Computing Systems*, 63(2):344–365, 2019. doi:10.1007/s00224-017-9772-y.
14. Armando Castañeda and Sergio Rajsbaum. New combinatorial topology bounds for renaming: the lower bound. *Distributed Computing*, 22(5-6):287–301, 2010. doi:10.1007/s00446-010-0108-2.
15. Armando Castañeda and Sergio Rajsbaum. New combinatorial topology bounds for renaming: the upper bound. *Journal of the ACM*, 59(1):3:1–3:49, 2012. doi:10.1145/2108242.2108245.
16. Armando Castañeda, Michel Raynal, and Julien Stainer. When and how process groups can be used to reduce the renaming space. In *16th International Conference on the Principles of Distributed Systems (OPODIS)*, LNCS 7702, pages 91–105. Springer, 2012. doi:10.1007/978-3-642-35476-2_7.

17 Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, July 1986. doi:10.1016/S0019-9958(86)80023-7.

18 Carole Delporte-Gallet, Hugues Fauconnier, Pierre Fraigniaud, and Mikaël Rabie. Distributed computing in the asynchronous LOCAL model. In *21st International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, LNCS 11914, pages 105–110. Springer, 2019. doi:10.1007/978-3-030-34992-9_9.

19 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985. doi:10.1145/3149.214121.

20 Pierre Fraigniaud, Marc Heinrich, and Adrian Kosowski. Local conflict coloring. In *57th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 625–634, 2016. doi:10.1109/FOCS.2016.73.

21 Mohsen Ghaffari, Juho Hirvonen, Fabian Kuhn, and Yannic Maus. Improved distributed Δ-coloring. *Distributed Computing*, 34(4):239–258, 2021. doi:10.1007/s00446-021-00397-4.

22 Mohsen Ghaffari, Juho Hirvonen, Fabian Kuhn, Yannic Maus, Jukka Suomela, and Jara Uitto. Improved distributed degree splitting and edge coloring. *Distributed Computing*, 33(3-4):293–310, 2020. doi:10.1007/s00446-018-00346-8.

23 Magnús M. Halldórsson, Fabian Kuhn, Yannic Maus, and Tigran Tonoyan. Efficient randomized distributed coloring in CONGEST. In *53rd ACM Symposium on Theory of Computing (STOC)*, pages 1180–1193, 2021. doi:10.1145/3406325.3451089.

24 Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, November 1999. doi:10.1145/331524.331529.

25 Maurice Herlihy and Nir Shavit. On the Nature of Progress. In Antonio Fernàndez Anta, Giuseppe Lipari, and Matthieu Roy, editors, *Principles of Distributed Systems*, pages 313–328, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-25873-2_22.

26 Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992. doi:10.1137/0221015.

27 Yannic Maus. Distributed Graph Coloring Made Easy. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21, pages 362–372, New York, NY, USA, July 2021. Association for Computing Machinery. doi:10.1145/3409964.3461804.

28 Moni Naor and Larry J. Stockmeyer. What can be computed locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

29 David Peleg. *Distributed computing: a locality-sensitive approach.* Society for Industrial and Applied Mathematics, USA, 2000.

30 Václav Rozhon and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *52nd ACM Symposium on Theory of Computing (STOC)*, pages 350–363, 2020. doi:10.1145/3357713.3384298.

A Coloring General Graphs

It is possible to extend Algorithm 1 to connected graphs with maximum degree Δ, for every $\Delta \geqslant 2$ (see Algorithm 4). By construction, every process running Algorithm 4 returns a color taken in the set

$$\{\, (a, b) \mid a + b \leqslant \Delta \,\},$$

of cardinality $\frac{(\Delta+1)(\Delta+2)}{2} = \mathrm{O}(\Delta^2)$. The proof of correctness for Algorithm 4 uses the same arguments as for establishing the correctness of Algorithm 1. In particular, a process cannot run forever whenever its identifier becomes a local extremum among the identifiers of its active neighbors, which guarantee that every process eventually terminates.

Algorithm 4 $O(\Delta^2)$-coloring algorithm for general graphs, code for process p with neighbors $q_1, \ldots, q_k$, $k \leqslant \Delta$.

```
1 Input: X_p ∈ ℕ
2 Initially:
3     c_p = (a_p, b_p) ← (0, 0) ∈ ℕ × ℕ
4 Forever:
5     write(X_p, c_p) and read((X_{q_1}, c_{q_1}), ..., (X_{q_k}, c_{q_k}))  ▷ immediate snapshot
6     if c_p ∉ { c_{q_1}, ..., c_{q_k} } then return(c_p)
7     else
8         a_p ← min ℕ ∖ { a_u | u ∼ p, X_u > X_p }
9         b_p ← min ℕ ∖ { b_u | u ∼ p, X_u < X_p }
```

B Technical proofs

B.1 Proofs of Section 3

Proof of Lemma 8. We proceed by induction on $t \in \mathbb{N}$. For $t = 0$, the claim is vacuously true, as $A_p(t) = B_p(t) = \varnothing$. For the inductive step, we suppose the claim holds for $t = 0, \ldots, T$, and we show that it holds for $t = T + 1$. If $p \notin \overline{\sigma}(T + 1)$ then we have $\widehat{X}_p(T + 1) = \widehat{X}_p(T)$ and $A_p(T + 1) = A_p(T)$, Thus the claim holds by induction. Let us assume that $p \in \overline{\sigma}(T + 1)$, and let $x \in A_p(t)$. By Equation (3) and the assumption $p \in \overline{\sigma}(T + 1)$, there exists $q \in N_p^+(T + 1)$ such that either $x = \widehat{X}_q(T + 1)$ or $x \in \widehat{A}_q(T + 1)$. In the former case, we have $\widehat{X}_p(T + 1) < Y$ by the definition of N_p^+. In the latter case, there must exist some time $t' \leqslant T$, with $q \in \overline{\sigma}(t')$, for which $\widehat{X}_q(T + 1) = X_q(t')$ and $\widehat{A}_q(T + 1) = A_q(t')$. Since $\tau \leqslant T$, we get that that $\widehat{X}_q(t') < x$, thanks to the induction hypothesis. Also, since the value of $X_q(t)$ is stable throughout the execution, we have $X_q(t') = X_q(t' - 1) = \widehat{X}_q(t') < x$. Therefore $\widehat{X}_q(T + 1) < x$, and, since $q \in N_p^+(T + 1)$, we have $\widehat{X}_p(T + 1) < \widehat{X}_q(T + 1) < x$, which proves the claim.

The proof is symmetric for $x \in B_p(t)$. ◀

Proof of Lemma 7. We establish the result for the case where X_p is not a local minimum and $a_p(t_1) = a_p(t_2) = a_p(t_3) = a_p(t_4)$. The proof uses the same arguments with local maxima and $b_p(t_1) = b_p(t_2) = b_p(t_3) = b_p(t_4)$.

Suppose that process p fails to return at time t_1; we consider two cases.

If p is a local maximum, then we have $\widehat{a}_p(t) = 0$ for all t. Moreover, if some process $q \sim p$ is working in the interval $[t_1, t_3]$, then $a_q(t_3) \neq 0$. Furthermore, we have $c_p(t_3) \neq \widehat{c}_q(t_3)$ by Lemma 5. In this case, either process q works in the interval $[t_3 + 1, t_4]$, and $\widehat{a}_q(t_4) \neq 0$, or it does not work in this interval, and $\widehat{c}_q(t_4) = \widehat{c}_q(t_3)$. Either way, we get $\widehat{c}_p(t_4) \neq \widehat{c}_q(t_4)$, and thus p returns at time at most t_4. Suppose now process p has a neighbor q' that is inactive in the interval $[t_1, t_3]$. If p's other neighbor q is *not* working in the interval $[t_1 + 1, t_2]$, then p returns at time t_2 by Lemma 6; if, on the other hand, q is working in this interval, then we have $a_q(t_2) \neq 0$, and, as above, process p returns at time at most t_3.

If process p is *not* a local maximum, then it has a neighbor q' with $X_{q'} > X_p$. If we had $\widehat{a}_p(t) = \widehat{a}_{q'}(t)$, in any $t \in \{ t_2, t_3, t_4 \}$, then $a_p(t)$ would be switched, which contradicts the lemma assumptions; hence we have $\widehat{c}_p(t) \neq \widehat{c}_{q'}(t)$ for $t = t_2, t_3, t_4$. Suppose p fails to return in t_2. In this case, as before, either the other neighbor q of p is working in the interval $[t_2 + 1, t_3]$, and so $a_q(t_3) \neq \widehat{a}_p(t_4)$; or q is inactive in that interval and p returns at time t_3. Either way, process p returns at the latest at time t_4. ◀

Proof of Lemma 10. We only treat the case $\left|N_p^+(t)\right| \leqslant 1$, as the other case is symmetric. First, note that for any process q, $\widehat{X}_q(t)$ is equal to either $\perp$ or X_q. In the former case, process q is still inactive in time t, and thus $A_q(t) = \varnothing$. As a consequence, thanks to Lemma 8, we have $X_q \notin A_q(t)$ for all $t \in \mathbb{N}$.

Given $p \in \overline{\sigma}(t)$, we proceed by induction over $|A_p(t)|$ by treating two base cases $|A_p(t)| = 0$, $|A_p(t)| = 1$, and then the general case. For the base cases, as $p \in \overline{\sigma}(t)$, we have $|A_p(t)| = 0$ if and only if $N_p^+(t) = \varnothing$, which corresponds to p being a local maximum among its neighbors awaken at time t. In this case, if p fails to return, then the algorithm enforces $a_p(t) = 0$, as desired, which gives the base case of the induction. If $|A_p(t)| = 1$, then the set $N_p^+(t)$ is a singleton. Let $\{ q \} = N_p^+(t)$. We have $A_p(t) = \{ \widehat{X}_q(t) \} = \{ X_q \}$, and $\widehat{A}_q(t) \in \{ \varnothing, \{ X_q \} \}$. The set $\widehat{A}_q(t)$ is therefore empty, i.e., $\widehat{a}_q(t) = 0$, and thus the algorithm enforces $a_p(t) = 1 = |A_p(t)|$.

For the inductive case, let us assume that the claim is true for $|A_p(t)| = 0, \dots, T$ with $T \geqslant 1$, and let us show that it still holds for $|A_p(t)| = T + 1 \geqslant 2$. Here again, the set N_p^+ has to be a singleton, say $\{ q \}$, and so we have $a_p(t) = 1 - \widehat{a}_q(t)$, and $A_p(t) = \{ X_q \} \cup \widehat{A}_q(t)$, with $X_q \notin \widehat{A}_q(t)$. Thus $\left|\widehat{A}_q(t)\right| = T$, and there was an earlier time $t' < t$ where $q \in \overline{\sigma}(t')$ failed to return, and $\widehat{A}_q(t) = A_q(t')$. Since $X_p < X_q$, $|N_q^+(t')| \neq 2$, and so $a_q(t') \equiv T \pmod 2$ by the induction hypothesis. Thus $a_p(t) = 1 - a_q(t') \equiv T + 1 \pmod 2$, which completes the proof of the claim. ◀

Proof of Lemma 16. We first show the following: if $p \in \overline{\sigma}(t)$ fails to return at time $t \geqslant 1$, then

$$a_p(t) = 0 \iff |A_p(t)| \equiv 0 \bmod 2. \tag{6}$$

We proceed by induction on $|A_p(t)|$. If $|A_p(t)| = 0$, then process p is a local maximum among its active neighbors, and so in Algorithm 2 we have $C^+ \leftarrow \varnothing$, which implies $a_p(t) = 0$. For the inductive step, suppose the result true for $|A_p(t)| = k$, and suppose that $|A_p(t)| = k + 1$. Since process p is assumed to be non-minimal, it has one neighbor q with $\widehat{X}_q(t) > \widehat{X}_p(t)$ and $|\widehat{A}_q(t)| = k$, and we have $a_p(t) = \min \mathbb{N} \setminus \{ \widehat{a}_q(t), \widehat{b}_q(t) \}$.

If k is even, then by inductive assumption we have $\widehat{a}_q(t) = 0$, and so $a_p(t) \neq 0$. Otherwise, k is odd, and by inductive assumption we have $\widehat{a}_q(t) > 0$. In the code of Algorithm 2, we have $C^+ \subseteq C$, and so for any process $u \in [n]$ and time $\tau \geqslant 0$ we have $b_u(\tau) \geqslant a_u(\tau)$. Thus in particular we have $\widehat{b}_q(t) \geqslant \widehat{a}_q(t) > 0$, and therefore $a_p(t) = 0$.

For the main claim, let $\ell \coloneqq |A_p(t_1)| = |A_p(t_2)| = |A_p(t_3)| = |A_p(t_4)|$. If ℓ is even, then $a_p(t) = 0$ for all $t \in [t_1, t_4]$. Reasoning as in Lemma 7, if p still hasn't returned by time t_4, then we have $|A_p(t_3)| = |A_q(t_3)| - 1 = |A_{q'}(t_3)| + 1$ without loss of generality. Then if neither q nor q' is activated in the interval $[t_3 + 1, t_4]$, p terminates by Lemma 6. Otherwise, using again the fact that $b_u(\tau) \geqslant a_u(\tau)$ for any process u and time τ, we have $\widehat{a}_p(t_4) = 0 < \min\{ \widehat{a}_q(t_4), \widehat{b}_q(t_4), \widehat{a}_{q'}(t_4), \widehat{b}_{q'}(t_4) \}$, and so p returns in t_4.

If ℓ is odd, we suppose without loss of generality that $X_q > X_p > X_{q'}$. We have $\widehat{a}_p(\tau) > 0$ for all $\tau \in [t_2, t_4]$, and reasoning again as in Lemma 7, by time t_4 we have $\widehat{a}_{q'}(t_4) = 0$, and p terminates if it is still active. ◀

B.2 Proofs of Section 4

Proof of Lemma 22. We show the following: for every $t \in \mathbb{N}$, $X_p(t) \notin \{ X_q(t), \widehat{X}_q(t) \}$, proceeding by induction. The case $t = 0$ results from the initial proper coloring of the identifiers.

For the induction, suppose the claim holds for $t = 0, \dots, T$. If $p, q \notin \overline{\sigma}(T + 1)$, then nothing changed, and the claim still holds for $t = T + 1$.

Suppose $p, q \in \overline{\sigma}(T+1)$. If $r_q(T+1) = r_q(T)$, the claim immediately follows, as does it if $X_q(T+1) = X_q(T)$. Otherwise, by assumption we either have $X_q(T) > X_p(T)$, or the opposite. If the former, $X_q(T+1) = f(X_q(T), X_p(T)) < X_p(T)$, and by Lemma 20 we have $X_q(T{+}1) \notin \{ X_p(T), X_p(T{+}1) \}$, and so $X_p(T{+}1) \notin \{ \widehat{X}_q(T{+}1), X_q(T{+}1) \}$. Otherwise, $X_q(T) < X_p(T)$; if q is a local minimum in $T+1$, then $X_q(T+1) \neq f(X_p(T), X_q(T))$, and the claim follows from $X_p(T+1) \in \{ X_p(T), f(X_p(T), X_q(T)) \}$. If q is *not* a local minimum, then $X_q(T+1) = f(X_q(T), Z) < Z$ for some $Z < X_q(T)$; here again, the claim follows from Lemma 20.

Finally, suppose $p \in \overline{\sigma}(T+1)$ and $q \notin \overline{\sigma}(T+1)$. If $X_p(T+1) = X_p(T)$, then the claim still holds. Otherwise, we have $r_p(T) < r_p(T+1) \leqslant \infty$, and $X_p(T+1) < X_p(T)$. Process p is then not a local maximum in $T+1$, and the algorithm guarantees $X_p(T+1) < \widehat{X}_q(T+1)$. If $X_q(T+1) = \widehat{X}_q(T+1)$, and in particular if $r_q(T+1) = \widehat{r}_q(T+1)$, and the claim holds.

Suppose then that $r_q(T{+}1) < \widehat{r}_q(T{+}1)$, and let t_0 be the earliest time when $r_q(t_0) = r_q(T)$, such that $\widehat{r}_q(t_0) = \widehat{r}_q(T{+}1)$. Process q takes no steps in the interval $(t_0, T{+}1]$, and because r_q increases in t_0, we have $\widehat{r}_q(t_0) \leqslant \widehat{r}_p(t_0)$. Thus $\widehat{r}_q(T+1) \leqslant \widehat{r}_p(t_0) \leqslant \widehat{r}_p(T+1)$. Since r_p increases in $T+1$, we have $\widehat{r}_p(T+1) \leqslant \widehat{r}_q(T+1)$, and thus

$$\widehat{r}_q(t_0) = \widehat{r}_p(t_0) = \widehat{r}_q(T+1) = \widehat{r}_p(T+1),$$

i.e., $\widehat{r}_p(t)$ is constant for $t \in [t_0, T{+}1]$, and as a consequence, $\widehat{X}_p(T{+}1) = \widehat{X}_p(t_0)$. From here, we proceed as in the previous case: $X_q(T{+}1) = X_q(t_0)$ was computed with q seeing $\widehat{X}_p(t_0) = \widehat{X}_p(T+1)$, and, since $X_p(T+1)$ was computed with p seeing $\widehat{X}_q(T+1) = \widehat{X}_q(t_0)$, we have indeed $X_p(T+1) \notin \{ X_q(T+1), \widehat{X}_q(T+1) \}$. ◀

Proof of Lemma 24. Since $p \sim q$, and since p is blocked at time t_0, we have $\widehat{r}_p(t_0) = \widehat{r}_q(t_0) + 1$. Moreover, as $\widehat{X}_p(t_0) \geqslant 10$, the fact that $\widehat{X}_p(t_0)$ is not a local minimum means that $\widehat{r}_{q'}(t_0) \geqslant \widehat{r}_p(t_0)$. Otherwise, by Lemma 19, X_p would be smaller than $X_{q'}$. In particular, process p remains blocked as long as process q is itself blocked.

Now, suppose that $r_q(t_1) \neq \widehat{r}_q(t_1)$. As processes p, q are blocked until t_1, we have $\widehat{X}_q(t_1) = \widehat{X}_q(t_0)$ and $\widehat{X}_p(t_1) = \widehat{X}_p(t_0)$, so $\widehat{X}_q(t_1) > \widehat{X}_p(t_1)$, and q is not a local minimum in t_1. The case $r_q(t_1) = \infty$ thus corresponds to $\widehat{X}_p(t_0)$ being a local maximum at time t_1. If $r_q(t_1) < \infty$, then $r_q(t_1) = \widehat{r}_q(t_0){+}1$, and since $\widehat{X}_p(t_1) = \widehat{X}_p(t_0) \geqslant 10$, we get $X_q(t_1) = f(\widehat{X}_q(t_0), \widehat{X}_p(t_0)) < \widehat{X}_p(t_0)$ by Lemma 19.

Finally, suppose then that q is next activated at time t_2, and that p is activated at time $t \geqslant t_2$.Note that as long as q does not get activated again, p remains blocked, as it did not see the update on r_q. Moreover, as $X_q(t_1)$ is not a local maximum, $X_q(t_2) < \widehat{X}_p(t_1) = \widehat{X}_p(t_0)$. Then process p sees $\widehat{X}_p(t_0)$ to be a local maximum, and since it is no longer blocked by q we have $r_p(t) = \infty$. ◀

Proof of Lemma 25. Under the hypotheses of the lemma, processes $q_1, \ldots, q_k$ remain blocked throughout the time interval $[t_0, t_1 - 1]$, and we have $\widehat{X}_{q_0}(t) > \widehat{X}_{q_1}(t) > \cdots > \widehat{X}_{q_k}(t)$ whenever $t_0 \leqslant t < t_1$. By the same arguments as the ones used in the previous section for establishing Lemma 10, the sign of $a_{q_i}(t)$ is determined by the sign of the variables $a_{q_0}, \ldots, a_{q_{i-1}}$ throughout the time interval $[t_0, t]$. In particular, the sign of $\widehat{a}_{q_i}(t)$ stabilizes after q_i has been activated at most $3\,i$ times, and thus process i itself terminates after having been activated at most $(3i+1)$ times. ◀

Proof of Lemma 26. Let $t_1, t_2, \ldots$ be the consecutive steps taken by process p in the interval $[t_0, \infty)$, that is, for every $k \geqslant 1$, p is inactive during the whole time (t_k, t_{k+1}). Note that, since $\widehat{r}_q(t_0) = \infty$, we have $\widehat{a}_q(t) = 0$ for all $t \geqslant t_0$, and so $\widehat{a}_p(t) > 0$ for all $t \geqslant t_2$, as process q will remain a local maximum forever.

Pick $k \geqslant 2$. If p is *not* a local minimum at any point in $[t_k, t_{k+3}]$, then by Lemma 16 it terminates in t_{k+3} at the latest. Conversely, if p *is* a local minimum throughout the same interval, then we have repeatedly $\widehat{a}_p(t_i) \in \{\, \widehat{a}_{q'}(t_i), \widehat{b}_{q'}(t_i) \,\}$, $i = k, \ldots, k+3$. This implies that $\widehat{a}_{q'}$ is positive during that interval, otherwise p and q' would eventually stop having conflicts. By the same argument, process q' terminates in a constant number of activations, and so do process p. Therefore, for process p *not* to terminate the relative order of $\widehat{X}_p$ and $\widehat{X}_{q'}$ must switch every $\mathrm{O}(1)$ steps. Thus, every time p takes $\mathrm{O}(1)$ steps and fails to return, it must be the case that $\widehat{X}_p$ has decreased. As argued before, this can happen at most $\mathrm{O}(\log^* n)$ times before either $X_p \leqslant 10$ or $X_{q'} \leqslant 10$, at which point convergence happens in a bounded number of steps. ◀

Proof of Lemma 27. Since process p is blocked, a direct induction shows that p lies somewhere within a monotone chain of identifiers, as described in Lemma 25. That is, there is a chain of distinct adjacent processes

$$q_{-k-1} \sim q_{-k} \sim \cdots \sim q_{-1} \sim q_0 \sim q_1 \sim \cdots \sim q_\ell,$$

with $q_0 = p$, and $k, \ell \geqslant 0$, where, for every $i \in [-k, \ell]$, $\widehat{r}_{q_i}(t_0) = \widehat{r}_p(t_0) + i$, and either $\widehat{r}_{q_{-k-1}}(t_0) = \bot$ (in which case $k = \widehat{r}_p(t_0)$) or $\widehat{r}_{q_{-k-1}}(t_0) = R - k - 1$ (in which case $k < \widehat{r}_p(t_0)$. Moreover, all processes $q_{-k}, \ldots, q_\ell$ are blocked at time t_0, and process q_{-k-1} is not blocked at time t_0.

We now distinguish two cases, depending on whether $k = 0$ or not. If $k = 0$, then, by Lemma 25, process p terminates after taking 4 steps within the time interval $[t_0, t_1)$. If $k > 0$, then process q_{-1} is blocked; if process q_{-1} remains blocked while process p takes $3k+1$ steps, then, by Lemma 25, p terminates. The same holds if q_{-1} takes a single non-blocked step. If process q_{-1} ever becomes unblocked, and takes another step, then we meet the assumptions of Lemma 24, and either of X_p or $X_{q_{-1}}$ become a local maximum. If X_p becomes a local maximum, then it terminates in $\mathrm{O}(1)$ steps. If $X_{q_{-1}}$ becomes a local maximum, then, by Lemma 26, process p terminates in $\mathrm{O}(\log^* n)$ steps. ◀

Distributed Randomness from Approximate Agreement

Luciano Freitas
LTCI, Télécom Paris, Institut Polytechnique de Paris, France

Petr Kuznetsov
LTCI, Télécom Paris, Institut Polytechnique de Paris, France

Andrei Tonkikh
LTCI, Télécom Paris, Institut Polytechnique de Paris, France

Abstract

Randomisation is a critical tool in designing distributed systems. The *common coin* primitive, enabling the system members to agree on an unpredictable random number, has proven to be particularly useful. We observe, however, that it is impossible to implement a truly random common coin protocol in a fault-prone asynchronous system.

To circumvent this impossibility, we introduce two relaxations of the perfect common coin: (1) *approximate common coin* generating random numbers that are *close* to each other; and (2) *Monte Carlo common coin* generating a common random number with an arbitrarily small, but non-zero, probability of failure. Building atop the *approximate agreement* primitive, we obtain efficient asynchronous implementations of the two abstractions, tolerating up to one third of Byzantine processes. Our protocols do not assume trusted setup or public key infrastructure and converge to the perfect coin exponentially fast in the protocol running time.

By plugging one of our protocols for *Monte Carlo common coin* in a well-known consensus algorithm, we manage to get a *binary Byzantine agreement* protocol with $O(n^3 \log n)$ communication complexity, resilient against an adaptive adversary, and tolerating the optimal number $f < n/3$ of failures without trusted setup or PKI. To the best of our knowledge, the best communication complexity for binary Byzantine agreement achieved so far in this setting is $O(n^4)$. We also show how the *approximate common coin*, combined with a variant of Gray code, can be used to solve an interesting problem of Intersecting Random Subsets, which we introduce in this paper.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Asynchronous, approximate agreement, weak common coin, consensus, Byzantine agreement

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.24

Related Version *Full Version*: https://arxiv.org/abs/2205.11878 [18]

Funding *Luciano Freitas*: Nomadic Labs.
Petr Kuznetsov: TrustShare Innovation Chair.
Andrei Tonkikh: TrustShare Innovation Chair.

1 Introduction

Generating randomness in distributed systems is an essential part of many protocols, such as Byzantine Agreement [4], Distributed Key Generation [21] or Leader Election [35]. Any application that needs an unpredictable or unbiased result will most likely rely on randomness. Although sometimes local sources of randomness are enough for some protocols [34], having access to a common random number can guarantee faster termination [3]. Producing a common unpredictable random number has been extensively studied in the literature on cryptography and distributed systems under the names of *random beacon*, distributed (multi-party) *random number generation* or *common coin* (even if the result is not binary). In essence, these protocols ensure:

© Luciano Freitas, Petr Kuznetsov, and Andrei Tonkikh;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 24; pp. 24:1–24:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Termination: every correct process eventually outputs some value;
Agreement: no two correct processes output different values;
Randomness: the value output by a correct process must be uniformly distributed over some domain $\mathcal{D}$, $|\mathcal{D}| \geq 2$.

We call a protocol that ensures the three properties (*Termination*, *Agreement*, and *Randomness*) a **perfect common coin**. There are many message-passing protocols without trusted setup that implement a perfect common coin in the presence of Byzantine adversary [40, 11, 25, 9, 7, 29, 19]. These protocols are either *synchronous*, meaning that every message sent by a correct process is delivered within a certain (known *a priori*) bound of time, or *partially-synchronous*, meaning that such a bound exists but is unknown.

In contrast, one can also consider an *asynchronous* system, where no bounds on communication delays can be assumed. In a seminal work of Fischer, Lynch, and Paterson, it has been shown that the problem of *consensus* has no asynchronous fault-tolerant solutions [17]. As we show in Appendix A, this impossibility also holds for perfect common coins: no algorithm can implement a perfect common coin in a message-passing asynchronous system where at least one process might crash. Note that this statement cannot be proven by a simple black-box reduction from consensus to a perfect common coin and a reference to FLP [17]. Indeed, if such a reduction existed, the resulting protocol would have to always terminate in a bounded number of steps, even with unfavourable outputs of the black-box common coin. Hence, if we were to replace the common coin protocol by a protocol that always returns 0, it would still provide termination as well as all other properties of consensus, violating [17].

Note that this impossibility applies even to systems with *trusted setup*, such as the one assumed in [9]. Such protocols typically do not satisfy the Randomness property of a perfect common coin. The outputs of these protocols follow deterministically from the information received by the processes during the setup.

In light of the impossibility of a perfect coin, one might look for *relaxed* versions of the common-coin problem that allow asynchronous fault-tolerant solutions. For example, one can relax the Agreement property by only requiring the output to be common with some constant probability, which results in an abstraction sometimes called *weak common coin*[1] or *δ-matching common coin*. In this paper, we call this abstraction a **probabilistic common coin**, in order to avoid confusion with other relaxations we introduce. More precisely, probabilistic common coins replace the Agreement property above with *probabilistic δ-consistency*.

Probabilistic δ-Consistency: with probability at least δ, no two correct processes output different values.

We also introduce the concept of a *Monte Carlo common coin* which is a probabilistic common coin whose success rate δ can be parameterized as follows: the more rounds of the protocol are executed, the more reliable the outcome is. In our case δ starts at $\frac{2}{3}$ in the first round of the protocol and converges to 1 at an exponential rate in the number of rounds. In most probabilistic common coins, δ could be increased by decreasing the resilience level (the allowed fraction of Byzantine processes). However, to the best of our knowledge, this paper is the first to present an implementation of a Monte Carlo common coin that can achieve arbitrarily small (but non-zero) δ by increasing the running time of the protocol (Sections 5 and 6).

---

[1] Many weak common coin protocols such as the one in [10] also relax Randomness.

We also propose a novel, alternative relaxation of Agreement: instead of ensuring that the *same* output is produced (with some probability), we may require that the produced outputs are *close* to each other according to some metric. For this variant, we have to also slightly relax randomness so that only one correct process is guaranteed to obtain a truly random value. More precisely, assume a discrete range of possible outputs $[0..D-1]$, and let $d_q(x, y)$ denote the distance between x and y in the algebraic ring $\mathbb{Z}_q$.[2] The **Approximate common coin** abstraction then satisfies Termination and the following two properties:

Approximate ϵ-Consistency: if one correct process outputs value x and another correct process outputs y, then $d_D(x, y) \leq \lceil \epsilon D \rceil$, for a given parameter $\epsilon \in (0, 1]$;

One Process Randomness: the value output by *at least one* correct process must be uniformly distributed over the domain $[0..D-1]$.[3]

Our implementations of Monte Carlo and Approximate common coins build upon the abstraction of *Approximate Agreement* [15]. It appears that the abstraction perfectly matches the requirements exposed by our relaxed common-coin definitions: it naturally grasps the notion of outputs being close where the precision can be related to the execution time. Building atop existing asynchronous Byzantine fault-tolerant implementations [15, 1], we introduce and discuss an efficient implementation of the *bundled* version of this abstraction which is, intuitively, equivalent to n parallel instances of Approximate Agreement, but is much more efficient.

We discuss two applications of our protocols. First, we observe that our Monte Carlo common coin can be plugged into many existing Byzantine agreement protocols [6, 10, 13, 33]. This helps us to obtain a *binary Byzantine agreement* protocol with $O(n^3 \lambda \log n)$ communication complexity, where λ is the security parameter. The protocol exhibits optimal resilience of $f < n/3$, tolerates adaptive adversary, and assumes no trusted setup or PKI. In this setting, the best prior protocols for binary Byzantine agreement we are aware of have communication complexity of $O(n^4 \lambda)$ [28, 2].

We also introduce *Intersecting Random Subsets*, a new problem that can be used to asynchronously choose random committees with large intersections. Using elements of coding theory, namely Gray Codes [23, 36], we show how our Approximate common coin can be used to solve this problem without additional communication overhead.

We present our model definitions in Section 2 and describe the building blocks used in our constructions in Section 3. We present our protocols in Sections 4–6, provided with implementation details and complexity analysis. In Section 7, we show how the Monte Carlo common coin can be used in solving binary Byzantine agreement. In Section 8, we overview the related work and in Section 9, we conclude the paper.

We provide some of the necessary complementary material in the appendices and we refer the reader to the full version of the paper [18] for the remaining parts. In Appendix A, we prove the impossibility of an asynchronous perfect common coin. Appendix B introduces and shows how to solve the problem of Intersecting random subsets. In addition to the contents of this paper, the full version [18] contains the proof of correctness of our common coin constructions, it contains a modular presentation of the coin proposed by Canetti and Rabin in 1993 [10], and it also presents two implementations of Random Secret Draw, one of the major building blocks of our common coins.

[2] I.e., $d(x, y) = \min\{|x - y|, q - |x - y|\}$.

[3] With a small modification to our protocol, we can easily achieve $(f+1)$-Process Randomness instead on One Process Randomness. However, we do not know if it is possible to guarantee that all outputs of correct processes are random without relaxing other properties.

2 System Model

We consider a system of n processes able to communicate using reliable communication channels. Among the participants, at most $f < \frac{n}{3}$ are Byzantine and might display arbitrary behaviour.

We assume the *adaptive* adversarial model: up to f Byzantine processes are chosen by the adversary depending on the execution. A non-Byzantine process is called *correct*. The communication complexity of our baseline protocols can be improved by a factor of n using Aggregatable Publicly Verifiable Secret Sharing (APVSS) [24]. However, as we are not aware of APVSS implementations that are secure against the adaptive adversary, the improved protocols can only be proved correct in the presence of the static one.

The adversary can control the time the messages sent by correct processes take to arrive, as well as reorder them. However, it cannot drop a message sent by a correct process unless it corrupts this process before the message has arrived.

We assume that each process has access to a local random number generator that can be accessed as follows:

RandomInt(D): produces a uniformly distributed random integer number in the range $[0..D-1]$.

The proposed protocols as well as some of the building blocks rely on the use of cryptographic hash functions. The hash of an arbitrary string s is denoted $H(s)$ and has length λ that we call the *security parameter*. It is computationally infeasible to find two strings $s \neq s'$ such that $H(s) = H(s')$, as well as inverting a hash without knowing which input was used a priori.

We assume a computationally bounded adversary, so that it is incapable of breaking cryptographic primitives with all but negligible probability. However, since such a probability exists, we allow the properties of all our protocols as well as all building blocks to be violated with a negligible in λ probability.

3 Building Blocks

Our protocols make use of a wide range of building blocks. None of them are completely new, but some of them are modified according to our needs. In particular, we introduce the *Random Secret Draw* abstraction inspired by the ideas from [10] and [2] (Section 3.3). We also provide a *bundled* version of the *Approximate Agreement* [15, 1] abstraction (Section 3.5). In addition, we use *Byzantine Reliable Broadcast* [6, 14] (Section 3.1), *Asynchronous Verifiable Secret Sharing* [8, 14] (Section 3.2), and *Gather* [10, 39] (Section 3.4).

3.1 Byzantine Reliable Broadcast

Byzantine Reliable Broadcast (BRB) [6] allows a designated leader to communicate a single message to all processes in such a way that, if any correct process delivers a message, then every other correct process eventually delivers exactly the same message (even if the leader is Byzantine). More precisely, a BRB protocol must satisfy the following properties:

Validity: if the leader is correct and it broadcasts message m, then every correct process will eventually deliver m;

Consistency: if two correct processes j and k deliver messages m_j and m_k, then $m_j = m_k$.

Totality: if a correct process j delivers some message m, then eventually all correct processes will deliver m.

The performance of reliable broadcast is of crucial importance to our protocols. We believe that the BRB implementation recently proposed by Das, Xiang, and Ren [14] will be the most suitable option. It has total communication complexity of just $O(n|M| + n^2\lambda)$, where $|M|$ is the size of the message and λ is the security parameter, total message complexity of $O(n^2)$, and the latency of 3 message delays in case of a correct leader and 4 message delays in case of a Byzantine leader.

In this paper, we always use BRB in groups of n instances, with each process being the leader of one. We use the following notation:

BRB$_i$.Broadcast(m): allows process i to broadcast a message in an instance of BRB where i is the leader;

BRB$_i$.Deliver(m): an event indicating that message m from process i has been delivered.

3.2 Asynchronous Verifiable Secret Sharing

Asynchronous Verifiable Secret Sharing (AVSS) [8] allows a process to securely share information with other participants and to keep its contents secret until the moment a threshold of participants agree to open it.

In our protocols, AVSS is used with the following interface:

AVSS$_i$.ShareSecret(x): allows process i to share a secret x among the participants;

AVSS$_i$.SharingComplete$()$: an event issued when a secret is correctly shared by process i;

AVSS$_i$.EnableRetrieve$()$: enables responses to retrieval requests;

AVSS$_i$.Retrieve$()$: returns x if it was previously shared and all correct processes invoked AVSS$_i$.EnableRetrieve().

An AVSS implementation must satisfy the following properties:

Validity: if a correct process i invokes AVSS$_i$.ShareSecret(x), then every correct process eventually receives the AVSS$_i$.SharingComplete() event and no value other than x can be returned from the AVSS$_i$.Retrieve() operation invoked by a correct process;

Notification Totality: if one correct process receives the AVSS$_i$.SharingComplete() event, then every correct process eventually receives it;

Retrieve Termination: if all correct processes invoke AVSS$_i$.EnableRetrieve() and any correct process invokes AVSS$_i$.Retrieve(), then this operation will eventually terminate and the process will obtain the shared secret;

Binding: if some correct process receives the AVSS$_i$.SharingComplete() notification, then there exists a fixed secret x such that no value other than x can be returned from the AVSS$_i$.Retrieve() operation invoked by a correct process;

Secrecy: if process i is correct and no correct process invoked AVSS$_i$.EnableRetrieve(), then the adversary has no information about the secret shared by i.

Das, Xiang, and Ren [14] proposed an AVSS protocol with quadratic communication complexity, constant latency, and without assuming trusted setup. Notice that in order to secretly share a long string s, it is better to follow the method proposed in [30]: encrypt s using a much shorter secret key *sym*, reliably broadcast the encrypted value $\{s\}_{sym}$ and then perform secret sharing of the key *sym*. Thus, we shall assume that the total communication complexity of secret sharing of string s is $O(n|s| + n^2\lambda)$.

3.3 Random Secret Draw

One of the key ideas of the weak common coin protocol of Canetti and Rabin [10] is to *assign* each process a random number in a given domain $[0..D-1]$ in such a way that:

Assignment Termination: if a correct process i participates, then it is eventually assigned a value. Moreover, everyone will eventually receive a notification that i has been assigned a random value;

Notification Totality: if process i receives a notification that some process j has been assigned a value, then every correct process will eventually receive such a notification;

Randomness: the assigned numbers are independent and distributed uniformly over the domain $[0..D-1]$. The distribution of the value assigned to process j cannot be affected by the adversary even if j itself is Byzantine;

Unpredictability: until at least one correct process agrees to reveal the assigned values, the value assigned to each process j remains secret, even to process j itself;

Retrieve Termination: if all correct processes invoke EnableRetrieve() and any correct process invokes RetrieveValue(j), having received ValueAssigned(j), then this operation eventually returns a value.

Although this idea has been widely used as part of the implementation of asynchronous consensus protocols, to the best of our knowledge, it was never considered a separate primitive and assigned a name. Hence, we shall call it *Random Secret Draw (RSD)*.

This abstraction resembles a well known concept of a Verifiable Random Function (VRF) [32]. However, the important difference is that process j itself cannot know the value it is assigned until the reveal phase. Hence, a Byzantine process cannot choose whether it wants to participate or not based on the random value it is assigned. Moreover, unlike Random Secret Draw, VRF schemes typically require a seed chosen at random *after* the process chose the public key for its pseudo-random function. In fact, a variant of RSD has been recently used to generate such seeds [20].

We use the following interface for the RSD abstraction:

RSD.Start(): allows a process to start participating in RSD and, eventually, to be assigned a random number. We assume that this function is non-blocking, i.e., that an invocation of this function terminates after 0 message delays;

RSD.EnableRetrieve(): used by the processes to start participating in the process of reconstructing the assigned values;

RSD.RetrieveValue(j): returns the value assigned to process j if all correct processes invoked RSD.EnableRetrieve() and process j has been assigned some value.

The original RSD implementation by Canetti and Rabin [10] used n^2 instances of AVSS. To the best of our knowledge, to this day, there is no known AVSS protocol that would allow to do it with less than $\Omega(n^4)$ bits of communication in total. Hence, in the full version of the paper [18], we give two possible implementations of RSD. The first one is secure against an adaptive adversary and does not rely on PKI, while the second one uses the implementation from [2] that relies on *Aggregatable Publicly Verifiable Secret Sharing* instead of AVSS. While saving a linear factor in communication complexity, this solution lacks security against adaptive adversary and requires PKI. Since both [10] and [2] did not consider RSD as a separate abstraction and did not provide separate pseudocode for it, in [18], we present both RSD implementations.

3.4 Gather

Yet another important contribution made by Canetti and Rabin in their weak common coin construction [10] is a multi-broadcast protocol that has been recently given the name *Gather* [39, 2]. In this protocol, every process starts by broadcasting a single message through Byzantine Reliable Broadcast. The processes then do a few more rounds of message exchanges

and, in the end, each participant i outputs a set of process ids $\mathcal{S}_i$ such that for all $j \in \mathcal{S}_i$: i has received the message of j through reliable broadcast.[4] Moreover, the sets output by correct processes satisfy a strong intersection property:

Binding Common Core: There exists a set $\mathcal{S}^*$ of process ids of size at least $n - f$, called the *common core*, such that for every correct process i: $\mathcal{S}^* \subseteq \mathcal{S}_i$. Moreover, once the first correct process outputs, $\mathcal{S}^*$ is fixed and the adversary cannot manipulate it anymore.

The fact that the adversary cannot affect the common core once a single correct process outputs will be important in our protocols. The adversary should not be able to choose the common core based on the generated random numbers after some of the correct processes invoked EnableRetrieve.

We slightly generalize the interface of Gather by using it in conjunction with BRB, but also with other similar primitives (in particular, AVSS and RSD) and their combinations. When a process invokes GATHER, it passes to it an arbitrary callable function GatherAccept that takes a process id j and returns *true* if the message from this process is considered to be delivered (not necessarily through BRB). We assume that Gather exports the following interface:

Gather.Start(GatherAccept): allows a process to start participating in the Gather protocol;

Gather.DeliverSet(S): provides the output of the Gather protocol.

In order for the protocol to terminate, the GatherAccept function has to satisfy properties similar to those of reliable broadcast.

Accept Validity: if a correct process i invoked GATHER.Start, then for every correct process j, GatherAccept(i) invoked by process j must eventually return *true*. Moreover, for all i, once GatherAccept(i) returned to *true* to some correct process, it must never switch back to *false*;

Accept Totality: if GatherAccept(i) invoked by one correct process returned *true*, then eventually it must return *true* to all correct processes.

Thanks to the properties of AVSS and Random Secret Draw (in particular, to the Notification Totality property), in our protocols, this assumption is trivially satisfied. For Gather, we use the original protocol of [10] (to the best of our knowledge, it was first described as a separate primitive in [39]).

3.5 Bundled Approximate Agreement

The last building block that we shall need is *Approximate Agreement (AA)* [16]. In a (one-dimensional) AA instance, the processes propose inputs and produce outputs (real values) so that the following properties are satisfied:

Validity: the outputs of correct processes must be in the range of inputs of correct processes.

Approximate $\tilde{\epsilon}$-consistency: the values decided by non-faulty processes must be at most a distance $\tilde{\epsilon}$ apart from each other.[5]

Termination: every non-faulty process eventually decides.

In our algorithms, Approximate Agreement is always executed in *bundles* of n parallel instances. For the sake of efficiency, one can treat it as a bundled problem with an input vector of size n, corresponding to the different instances and then, for every message, send

[4] In [39] and [2], Gather returns a set of pairs $(id, value)$. However, for our purposes, working with sets of ids is more convenient. The values will be delivered through normal BRB.Deliver event.

[5] We use $\tilde{\epsilon}$ to distinguish it from ϵ used in the definition of Approximate Common Coin.

information about all instances at the same time, but treat them separately as before. We call this abstraction *Bundled Approximate Agreement* (BAA). BAA **should not be confused** with *Multidimensional Approximate Agreement* [31], which is a stronger abstraction than the one we rely upon.

Assuming binary inputs, the processes access BAA via the following interface:

BAA.Run($[x_1, x_2, \ldots, x_n]$): Launches n instances of Approximate Agreement protocol, where the input for the i-th instance is x_i. For a given parameter $\tilde{\epsilon}$, the protocol is executed until $\tilde{\epsilon}$-approximation is satisfied in every instance and then returns a vector of outputs $[y_1, y_2, \ldots, y_n]$.

For implementing BAA, we suggest using the Approximate Agreement protocol proposed in [1] with resilience $f < \frac{n}{3}$. Since in our protocols, the inputs are either 0 or 1, we do not need the termination detection techniques described in [1] neither do we need the "init" phase of the protocol. With the aforementioned *BRB* and some trivial changes[6], this implementation will give us the communication complexity of $O(n^3\lambda)$ and latency $4 \cdot \log_2(1/\tilde{\epsilon})$.

4 Approximate Common Coin

Algorithm 1 Approximate common coin.

```
1:  Instance parameters: domain D, precision ε

2:  Distributed objects:
3:      ∀j ∈ [n] : AVSS_j – instance of Asynchronous Verifiable Secret Sharing with leader j
4:      GATHER – instance of Gather
5:      BAA – instance of Bundled Approximate Agreement with precision ε̃ = ε/f

6:  function GatherAccept(j)
7:      return true if received AVSS_j.SharingComplete()

8:  operation Toss() returns integer
9:      x = RandomInt(D)
10:     AVSS_i.ShareSecret(x)
11:     GATHER.Start(GatherAccept)

12:     wait for GATHER.DeliverSet(S)
13:     ∀j ∈ [n] : let w_j = { 1, j ∈ S,
                              0, otherwise
14:     [w'_1, ..., w'_n] = BAA.Run([w_1, ..., w_n])
15:     ∀j ∈ [n] : AVSS_j.EnableRetrieve()      // only after BAA completes
16:     ∀j ∈ [n]: let x_j = { AVSS_j.Retrieve(),  if w'_j ≠ 0
                             0,                   otherwise

17:     // Compute and return the final random number
18:     return (⌈∑_{j∈[n]} x_j · w'_j⌉ mod D)
```

The main idea of this protocol is to aggregate numbers locally generated by enough different processes so that at least one of them is correct and the number it generated is truly random and uniformly distributed. With a good aggregation function, the resulting value will also be uniformly distributed. An example of such an aggregation function is addition modulo the size of the domain. Indeed, it is easy to see that, if x is uniformly distributed

[6] In the "report" messages, hashes of the values should be sent instead of the values themselves.

over $[0..q-1]$ and y is any number chosen independently of x, then $(x+y) \bmod q$ will also be uniformly distributed over $[0..q-1]$. Another example of an aggregation operation that satisfies a similar property is bit-wise xor (as long as the domain size is a power of 2).

However, without being able to solve consensus, we cannot just elect $f+1$ or $n-f$ processes from whom we shall take these values. Thankfully, unlike xor, addition has one more useful property: it is continuous. If we take two numbers, x and y, such that $d_q(x,y) \leq \alpha$, then for any number z, $d_q(z+x, z+y) \leq \alpha$.[7] Hence, a natural idea is to *approximately* elect the set of processes to provide the random inputs.

More precisely, in order to produce an *approximate common coin* in the range $[0..D-1]$, each process locally generates and secret-shares a random number in this range. Then each process gathers a set of ids of processes that have completed the sharing (line 12).

The next step is to create a binary vector with n positions, where each position j is set to 1 if and only if j is present in the gathered set (line 13). This vector is then used as an input for the BAA protocol (line 14), which outputs a vector W of *weights* such that for each position j, the weights received by different processes are at most $\tilde{\epsilon}$ apart.

The value of $\tilde{\epsilon} = \epsilon/f$ is chosen such that the final outputs of the coin are at most ϵ apart from each other. For the details on how this particular value was computed, see [18].

Recall that BAA ensures that the output values lie within the range of of inputs of correct processes. Moreover, by the properties of Gather and AVSS, if at least one correct process has j in its gathered set, then j has correctly shared its value and it can be later retrieved by the correct processes. Therefore if the j-th value is irretrievable, the j-th component will always be assigned weight 0. On the other hand, due to the common core property of Gather, at least $n-f$ values will have weight 1, which guarantees that the result is uniformly distributed in the desired range.

Finally, the processes reveal the secrets (lines 15 and 16) and compute the resulting random number.

▶ **Theorem 1.** *Algorithm 1 implements an approximate common coin.*

The proof of Theorem 1 is presented in the full version of this paper [18].

Complexity analysis. The communication complexity of our *approximate common coin* can be broken down into:

1. n instances of AVSS.ShareSecret in parallel $\Rightarrow O(n^3\lambda)$;
2. One instance of GATHER $\Rightarrow O(n^3\lambda)$;
3. One instance of BAA with $\tilde{\epsilon} = \epsilon/f \Rightarrow O(n^3\lambda(\log f + \log \frac{1}{\epsilon}))$;
4. n instances of AVSS.Retrieve in parallel $\Rightarrow O(n^3\lambda)$;

Hence, the total communication complexity is $O(n^3\lambda(\log f + \log \frac{1}{\epsilon}))$ with Bundled Approximate Agreement being the bottleneck.

The time complexity of the protocol is $O(\log f + \log \frac{1}{\epsilon})$.

5 Monte Carlo Common Coin from Approximate Common Coin

In this section, we present a simple reduction from an approximate common coin to a Monte Carlo common coin. The very short pseudocode is in Algorithm 2.

[7] Recall that $d_q(x,y)$ is the distance between x and y in the ring $\mathbb{Z}_q$, i.e., $d_q(x,y) = \min\{|x-y|, q-|x-y|\}$.

Algorithm 2 Monte Carlo Common Coin from Approximate Common Coin, code for process i.

```
19: Instance parameters: domain size D, success probability δ
20: let k = ⌈2/(1−δ)⌉
21: Distributed objects:
22:     AC – instance of approximate common coin with domain size kD and precision ϵ = 1/(kD)
23: operation Toss() returns integer
24:     return ⌊AC.Toss() / k⌋
```

The transformation requires first to generate an approximate common coin of domain kD where k is an integer number $\lfloor\frac{2}{1-\delta}\rfloor$ and $\epsilon = \frac{1}{kD}$. This implies that different processes shall get values at most $\lceil\frac{1}{kD} \cdot kD\rceil = 1$ apart.

The domain of the approximate coin is k times larger than the domain of the targeted Monte Carlo common coin. By dividing the result by k, we get the desired range of values and success probability δ, where k values of the *approximate common coin* are mapped to one value of *Monte Carlo common coin.*

▶ **Theorem 2.** *Algorithm 2 implements a Monte Carlo common coin with domain D and success probability δ.*

Proof. Termination and Unpredictability follow from the properties of *approximate common coin*, while Randomness follows from One Process Randomness since exactly k values from the larger domain (kD) are mapped to each value in the smaller domain (D). Let x' be the resulting toss of the first correct process that completes BAA in its approximate common coin toss. Then, as established, every other process will be at a distance at most 1 from it. Hence, if the remainder of the division of x' by k is neither 0 nor $k-1$, every correct process decides the same value:

$$\Pr[\text{failure}] \leq \frac{2}{k} = \frac{2}{\lceil\frac{2}{1-\delta}\rceil} \leq 1-\delta.$$ ◀

Complexity analysis. This protocol runs a single instance of an Approximate Common Coin, with precision $\epsilon = \frac{1}{D\lfloor\frac{2}{1-\delta}\rfloor}$. If used with our algorithm from Section 4, it will take $O(\log f + \log\frac{1}{\epsilon}) = O(\log f + \log D + \log\frac{1}{1-\delta})$ rounds of approximate agreement.

Hence, the overall time complexity of the protocol is $O(\log f + \log D + \log\frac{1}{1-\delta})$ and the communication complexity is $O(n^3\lambda(\log f + \log D + \log\frac{1}{1-\delta}))$.

6 Direct Implementation of Monte Carlo Common Coin

Overview. The main idea of this protocol is to assign to each process a random value and a random ticket. Then, using approximate agreement, the protocol is able to select the process with maximum ticket with adjustable probability of success and adopt the value corresponding to this ticket as the coin value.

Tickets and values. The protocol first assigns random values and random tickets to each participant, maintaining both secret until later (lines 37 and 38). Processes then gather a list of participants who have both drawn a ticket and a value, guaranteeing that all processes will hold sets that all intersect in at least $n-f$ participants (line 39).

Algorithm 3 Monte Carlo Common Coin, code for process i.

```
25: Instance parameters: domain size D, success probability δ, security parameter λ

26: Functions:
27:     Calibrate(w) – returns the weight to apply to a ticket given that BAA returned w

28: Distributed objects:
29:     TicketDraw – instance of Random Secret Draw with domain size 2^λ
30:     ValueDraw – instance of Random Secret Draw with domain size D
31:     Gather – instance of Gather
32:     BAA – instance of Bundled Approximate Agreement with precision ε̃,
33:             where ε̃ depends on the calibration function (see "Weight calibration" below)

34: function GatherAccept(j) returns boolean
35:     return true iff received both TicketDraw.ValueAssigned(j) and ValueDraw.ValueAssigned(j)

36: operation Toss() returns integer
37:     TicketDraw.Start()
38:     ValueDraw.Start()
39:     Gather.Start(GatherAccept)

40:     wait for event Gather.DeliverSet(S)
41:     ∀j ∈ [n] : let w_j = { 1, j ∈ S,  0, otherwise }
42:     [w'_1, ..., w'_n] = BAA.Run([w_1, ..., w_N])
43:     candidates = {j ∈ [n] | w'_j > 0}

44:     TicketDraw.EnableRetrieve()
45:     ValueDraw.EnableRetrieve()
46:     tickets = {TicketDraw.RetrieveValue(j) | j ∈ candidates}
47:     values = {ValueDraw.RetrieveValue(j) | j ∈ candidates}

48:     winner = argmax_{j ∈ candidates} Calibrate(w'_j) · tickets[j]
49:     return values[winner]
```

Approximate Agreement. In a similar manner as in the previous protocol, each process runs Bundled Approximate Agreement inputting 1 in the dimensions corresponding to the processes it has received from Gather, and 0 in the other dimensions (line 42). Similar to the previous protocol, if a process has not made a valid draw it will always be assigned weight zero, whereas if the weight is different than zero then it is possible to recover the secretly drawn number.

Opening the secrets. Prior to the first decision of a correct process in BAA, no secrets are leaked, as the underlying Random Secret Draw abstraction requires at least one correct process to invoke the EnableRetrieve operation before any information about the generated numbers is revealed. After this first decision of a correct process, the secrets can be opened (line 45), but at this point the adversary can only induce other processes deciding values which are at most $\tilde{\epsilon}$ apart from the first decision, which does not undermine the safety of the protocol.

Decision. With the tickets and values now openly available, the processes calibrate the tickets by multiplying the plain ticket by a *calibration function* applied to the weights. The simplest calibration function is an identify function, the calibrated ticket of a process i is simply the product of the output i-th output of BAA and the original ticket t_i. In their final steps, processes decide the value corresponding to the highest calibrated ticket (lines 48 and 49).

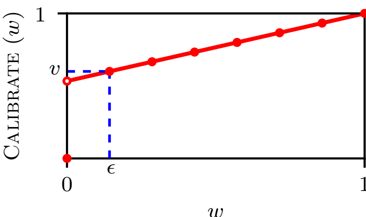


Figure 1 The weight calibration function.

Weight calibration. The protocol without weight calibration (i.e., with $\mathsf{Calibrate}(w) = w$) requires $3 + \lceil \log_2(n) + \log_2\left(\frac{1}{1-\delta}\right) \rceil$ rounds of approximate agreement in order to achieve the success probability δ. A similar performance is achieved by the protocol in Section 5.

In order to get rid of the $\log_2(n)$ part in the time complexity, we use a calibration function that is linear on $(0, 1]$ with a discontinuity point at 0, as illustrated in Figure 1. If $w_1 > 0$ and $w_2 > 0$ and $|w_1 - w_2| = \tilde{\epsilon}$, then, after calibration, $|\mathsf{Calibrate}(w_1) - \mathsf{Calibrate}(w_2)| \approx \tilde{\epsilon} \cdot (1 - v)$. This is similar in effect to running extra $\log_2 \frac{1}{1-v}$ rounds of approximate agreement, but at no extra latency cost. We then balance the value of the parameter v in such a way that, intuitively, the discontinuity at 0 is very unlikely to cause disagreement. An example of a good value for v that achieves this goal is $1 - \frac{\ln(2/(1-\delta))}{2n/3}$. In order to achieve success probability δ, $5 + \left\lceil \log_2\left(\frac{1}{1-\delta}\right) + \log_2\left(\log_2\left(\frac{1}{1-\delta}\right)\right) \right\rceil$ rounds of approximate agreement are required.

▶ **Theorem 3.** *Algorithm 3 implements a Monte Carlo common coin.*

The proof of Theorem 3 is presented in the full version of this paper [18].

Complexity analysis. The communication complexity of our *Monte Carlo common coin* can be broken down into:

1. 2 instances of Random Secret Draw $\Rightarrow O(n^3\lambda)$;
2. 1 instance of GATHER $\Rightarrow O(n^3\lambda)$
3. One instance of BAA with $\tilde{\epsilon} = O(1/(1-\delta)) \Rightarrow O(n^3\lambda \log \frac{1}{1-\delta})$;
4. $2n$ secret retrievals in parallel $\Rightarrow O(n^3\lambda)$;

Hence, the total communication complexity is $O(n^3\lambda \log \frac{1}{1-\delta})$ with Bundled Approximate Agreement being the bottleneck. The time complexity of the protocol is $O(\log \frac{1}{1-\delta})$.

7 Applications

Asynchronous Binary Byzantine Agreement. Monte Carlo common coins can be plugged into any Byzantine Agreement protocol that makes a call to a probabilistic common coin, such as [6, 10, 13, 33]. Given any probabilistic common coin with constant success probability, these algorithms terminate in a constant number of rounds, exchanging at most $O(n^3\lambda)$ bits.

Using our Monte Carlo common coin obtained via the transformation from approximate common coin (Section 5), we get a protocol that is secure against an adaptive adversary, assumes no trusted setup or PKI, and exhibits communication complexity $O(n^3\lambda \log n)$ at the expense of extra $O(\log n)$ factor in time complexity (the complexity of coin flips

dominate the rest of the protocol). As far as we know, the best existing setup-free solutions that are resilient against adaptive adversary and tolerate up to $f < n/3$ failures exhibit communication complexity of $O(n^4\lambda)$ [28, 2].

Intersecting Random Subsets. An example of an interesting new application that can be solved with Approximate Common Coins but not probabilistic common coins, with direct consequences into the choice of committees among processes, is given in Appendix B.

8 Related Work

Ben-Or [4] proposed the first randomized consensus algorithm based on the "independent choice" common coin. In this algorithm, every participant tosses a local random coin and with probability 2^{-n}, the values picked by n participants are identical. Bracha [6] extended this algorithm to the Byzantine fault model with $f < n/3$ faulty participants.

Rabin [37] proposed an implementation of a perfect common coin based on Shamir's secret sharing [38], assuming trusted setup (a trusted dealer distributes *a priori* a large number of secrets). Today, most protocols that allow trusted setup use the solution proposed by Cachin et al. [9] who have described a practical perfect common coin based on threshold pseudorandom functions (tPRF), assuming that a trusted dealer distributes a short tPRF key. These protocols use the pre-distributed randomness in a clever way to obtain multiple numbers that are computationally indistinguishable from random (much like a PRNG [5]). In contrast, our algorithms do not assume trusted setup.

In the setup-free context, Canetti and Rabin [10] proposed a weak common coin algorithm based on asynchronous verifiable secret sharing (AVSS), which resulted in an efficient randomized *binary* consensus. In designing our protocols, we make use of multiple ideas from this work, taking into account recent improvements, such as the use of Aggregatable Publicly Verifiable Secret Sharing [24], suggested in a similar context by Abraham et al. [2].

Using standard PKI cryptography, Cohen et al. [12] built two common coins relying on VRFs. The first one with resilience $f < (1/3 - \epsilon)$ achieves success rate $\delta = \frac{18\epsilon^2+24\epsilon-1}{6(1+6\epsilon)}$. The second coin involves sampling committees among the processes and also guarantees a constant success rate that depends on the system's resilience.

Kogias et al. [28] proposed a relaxed abstraction called *eventually perfect common coin*. They first build a weak distributed key generator (wDKG) which is a protocol that never terminates: each party outputs a sequence of candidate keys to be used for encryption and decryption with the property that they will eventually agree on a set of keys. This mechanism can replace the trusted setup in [9]. Moreover, it can be shown that the participants may disagree on the set of keys at most $f + 1$ times. The resulting coin eventually terminates with a perfect result, hence its name. In contrast, the challenge of our work was to devise (one-shot) unbiased common coins with provable termination.

Gao et al. [20] combined VRFs and AVSS to produce the first random coin which has $O(n^3\lambda)$ communication complexity. With the advent of new broadcast and APVSS implementations, the classic protocol of [10] gains the same complexity. They created a weak form of *Gather* called *core-set selection* in which $f + 1$ correct participants share at least $n - f$ VRFs coming from different processes. They then use AVSS to build the seeds for VRFs and whenever the highest VRF is in the common core, the nodes successfully agree in the coin outcome. Their protocol assumes the static adversary, but it can be made adaptive with a *relatively weak* form of trusted setup: a single common random number must be published after the public keys of the participants are fixed. In contrast, our protocols do not assume any form of trusted setup. Assuming static adversary, our protocols can achieve the same communication cost while additionally enabling parameterized success rate.

Our Monte Carlo coin from Section 6 was inspired by the Proposal Election protocol recently introduced by Abraham et al. [2]. Technically, it is not a common coin *per se* but it uses elements of it. In this protocol, every party inputs some externally valid value, and with a constant probability, all parties output the same value that was proposed by a non-faulty party chosen at random. Intuitively, the main contribution of the protocol in Section 6 is the use of approximate agreement to *amplify* the success probability.

In the *full information* model, without using cryptography, King and Saia [27] observed that the strength of the Byzantine adversary is in its anonymity, but it cannot bias the coin indefinitely without being detected. Even though their Byzantine agreement algorithm with polynomial expected time does allow the adversary to bias the coin, but amended this with statistical tests aiming at detecting this kind of deviation and evicting misbehaving participants. The resilience level of this algorithm is, however, orders of magnitude lower than n ($f < n/400$ in the best case). Huang et al. [26] recently extended this work to achieve the resilience of $f < n/4$. In contrast, our algorithms use cryptographic tools to produce unbiased (approximate) outputs and maintain optimal resilience $f < n/3$.

Monte Carlo protocols are randomized algorithms that have a fixed number of rounds and yield results that are correct with a given probability, while Las Vegas protocols always give the correct results but do not have a fixed number of rounds. Notice that Las Vegas algorithms must have a fixed probability of terminating every round and thus can be converted into Monte Carlo by stopping after a fixed number of rounds and deciding a random value if termination is not attained.

Such a transformation could be applied to [28], but since their latency is a function of $O(f)$, the resulting Monte Carlo common coin would also have a latency which is a function of f, while our solution is independent of this parameter. Another option would be to create a set of keys using [2] which could be run a fixed number of rounds and then to use [9]. Since their expected number of rounds is not a function of f, this transformation would have the same asymptotic complexity as ours, but it would include many unnecessary message delays from the consensus protocol, from the verifiable gather and other parts of their ADKG that are not present in a direct implementation such as ours.

9 Conclusion

In this paper, we suggest 2 new relations of the common coin primitive implementable in a fully asynchronous environment. We provide efficient implementations based on a range of novel techniques. Our protocols are the first use of approximate agreement to generate random numbers, which is used to keep decided values close in the approximate common coin protocol and to increase the probability of agreement in the direct implementation of the Monte Carlo common coin. Moreover, we also introduced elements of coding theory that were not previously applied to the distributed computing realm in the solution of what we called intersecting random subsets.

Further studies are necessary to explore the full potential of using these new abstractions in the design of distributed protocols and to understand the theoretical limits of their performance.

Tight performance analysis for the Monte Carlo common coin. In the full version of the paper, we proved that in order to achieve success probability δ, our probably common coin protocol requires $3 + \lceil \log_2(n) + \log_2\left(\frac{1}{1-\delta}\right) \rceil$ or $5 + \left\lceil \log_2\left(\frac{1}{1-\delta}\right) + \log_2\left(\log_2\left(\frac{1}{1-\delta}\right)\right) \right\rceil$ rounds of approximate agreement, depending on whether weight calibration is used. While it seems to correctly reflect the asymptotic behaviour of the actual distribution, these bounds seem to be rather pessimistic when only a few rounds of approximate agreement are run.

For example, according to these bounds, with weight calibration, we will need 8 rounds in order to achieve $\delta = \frac{2}{3}$. However, in practice, $\delta = \frac{2}{3}$ is achieved with 0 rounds of approximate agreement as with probability at least 2/3 a process in the common core will have the largest ticket. As we estimated empirically through simulations, the actual δ that we obtain after running 8 rounds of approximate agreement is around 0.993 instead of 2/3.

Non-linear calibration function for the Monte Carlo common coin. Weight calibration is necessary to achieve the latency of $O(\log \frac{1}{1-\delta})$ rounds in our Monte Carlo common coin protocol. We chose a concrete linear function because it was relatively simple to analyze (as we could do a reduction to the case without the calibration). However, this function is unlikely to be optimal. The extra $\log_2(\log_2 \frac{1}{1-\delta})$ part in the resulting estimate on the number of rounds of approximate agreement is likely to be due to sub-optimal choice of the weight calibration function.

Approximate common coin without extra $\log_2(f)$ rounds. Using the magic of weight calibration, for Monte Carlo common coin, we managed to achieve $O(\log(1/\epsilon))$ time complexity, which is likely to be optimal. However, our approximate common coin protocol requires $\log_2 f + \log_2(1/\epsilon)$ rounds of approximate agreement and, hence, its time complexity depends on two variables: f and ϵ. In some applications, ϵ may be constant and $\log_2(f)$ can become the bottleneck.

Creating a protocol without this extra delay or proving a $\Omega(\log(f))$ lower bound would be an interesting contribution to the understanding of the approximate common coin abstraction. It would also mean that the transformation from approximate common to Monte Carlo common would result in more efficient coins of the latter type.

References

1 Ittai Abraham, Yonatan Amit, and Danny Dolev. Optimal resilience asynchronous approximate agreement. In *8th International Conference on Principles of Distributed Systems (OPODIS 2004)*, OPODIS'04, pages 229–239, Berlin, Heidelberg, 2004. Springer-Verlag. doi:10.1007/11516798_17.

2 Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching consensus for asynchronous distributed key generation. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 363–373, 2021.

3 Marcos K Aguilera and Sam Toueg. The correctness proof of ben-or's randomized consensus algorithm. *Distributed Computing*, 25(5):371–381, 2012.

4 Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *PODC '83: Proceedings of the annual ACM symposium on Principles of distributed computing*, pages 27–30, 1983.

5 Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudorandom bits. *SIAM journal on Computing*, 13(4):850–864, 1984.

6 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

7 Benedikt Bünz, Steven Goldfeder, and Joseph Bonneau. Proofs-of-delay and randomness beacons in ethereum. *IEEE Security and Privacy on the blockchain (IEEE S&B)*, 2017.

8 Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 88–97, 2002.

9 Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

10 Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 42–51, 1993.

11 Ignacio Cascudo and Bernardo David. Scrape: Scalable randomness attested by public entities. In *International Conference on Applied Cryptography and Network Security*, pages 537–556. Springer, 2017.

12 Shir Cohen, Idit Keidar, and Alexander Spiegelman. Not a COINcidence: Sub-Quadratic Asynchronous Byzantine Agreement WHP. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DISC.2020.25.

13 Tyler Crain. Two more algorithms for randomized signature-free asynchronous binary byzantine consensus with $t < n/3$ and $o(n^2)$ messages and $o(1)$ round expected termination, 2020. doi:10.48550/ARXIV.2002.08765.

14 Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous data dissemination and its applications. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2721, 2021.

15 D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, July 1986.

16 A.D. Fekete. Asynchronous approximate agreement. *Information and Computation*, 115(1):95–124, 1994. doi:10.1006/inco.1994.1094.

17 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

18 Luciano Freitas, Petr Kuznetsov, and Andrei Tonkikh. Distributed randomness from approximate agreement. *arXiv preprint*, 2022. arXiv:2205.11878.

19 Luciano Freitas de Souza, Andrei Tonkikh, Sara Tucci-Piergiovanni, Renaud Sirdey, Oana Stan, Nicolas Quero, and Petr Kuznetsov. Randsolomon: Optimally resilient random number generator with deterministic termination. In *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

20 Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Efficient asynchronous byzantine agreement without private setups. *arXiv preprint*, 2021. arXiv:2106.07831.

21 Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 295–310. Springer, 1999.

22 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 51–68, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132747.3132757.

23 Frank Gray. Pulse code communication. *United States Patent Number 2632058*, 1953.

24 Kobi Gurkan, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Aggregatable distributed key generation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 147–176. Springer, 2021.

25 Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY technology overview series, consensus system. *CoRR*, abs/1805.04548, 2018. arXiv:1805.04548.

26 Shang-En Huang, Seth Pettie, and Leqi Zhu. Byzantine agreement in polynomial time with near-optimal resilience. *arXiv preprint*, 2022. arXiv:2202.13452.

27 Valerie King and Jared Saia. Byzantine agreement in expected polynomial time. *J. ACM*, 63(2), March 2016. doi:10.1145/2837019.

28 Eleftherios Kokoris Kogias, Dahlia Malkhi, and Alexander Spiegelman. *Asynchronous Distributed Key Generation for Computationally-Secure Randomness, Consensus, and Threshold Signatures*, pages 1751–1767. Association for Computing Machinery, New York, NY, USA, 2020. doi:10.1145/3372297.3423364.

29 Mikhail Krasnoselskii, Grigorii Melnikov, and Yury Yanovich. No-dealer: Byzantine fault-tolerant random number generator. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 568–573. IEEE, 2020.

30 Hugo Krawczyk. Secret sharing made short. In *Annual international cryptology conference*, pages 136–146. Springer, 1993.

31 Hammurabi Mendes and Maurice Herlihy. Multidimensional approximate agreement in byzantine asynchronous systems. In *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '13, pages 391–400, New York, NY, USA, 2013. Association for Computing Machinery. `doi:10.1145/2488608.2488657`.

32 Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *40th annual symposium on foundations of computer science (cat. No. 99CB37039)*, pages 120–130. IEEE, 1999.

33 Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous binary byzantine consensus with t < n/3, o(n2) messages, and o(1) expected time. *J. ACM*, 62(4), September 2015. `doi:10.1145/2785953`.

34 Achour Mostefaoui, Matthieu Perrin, and Michel Raynal. A new insight into local coin-based randomized consensus. In *2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 207–20709. IEEE, 2019.

35 Achour Mostefaoui and Michel Raynal. Leader-based consensus. *Parallel Processing Letters*, 11(01):95–107, 2001.

36 Andre Neubauer, Jurgen Freudenberger, and Volker Kuhn. *Coding theory: algorithms, architectures and applications*. John Wiley & Sons, 2007.

37 Michael Rabin. Randomized Byzantine generals. In *Proceedings of the 24th Symposium on Foundations of Computer Science*, pages 403–409. IEEE Computer Society Press, November 1983.

38 Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

39 Gilad Stern and Ittai Abraham. Living with asynchrony: the gather protocol. URL: `https://decentralizedthoughts.github.io/2021-03-26-living-with-asynchrony-the-gather-protocol`, 2021. Accessed: 2022-02-12.

40 Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 444–460. Ieee, 2017.

A Impossibility of an Asynchronous Perfect Common Coin

A perfect common coin protocol makes sure that the correct processes agree on a random number taken uniformly over a specific domain.

By reusing the classical arguments of the impossibility of asynchronous consensus [17], we are going to show that no asynchronous perfect common coin protocol may exist, if a single process is allowed to fail by crashing. Recall that we consider protocols in which processes may non-deterministically choose its actions based on local coin tosses.

Formally, a protocol provides each process with an automaton that, given the process' state and an input (a received message and a result of a local random coin toss), produces an output (a finite set of messages to send and/or the application output). We assume that the automaton itself is deterministic, i.e., all non-determinism is delegated to the outcomes of local random coins. A step of process p is therefore a tuple (p, m, r), where m is the message p receives (can be $\perp$ if no message is received in this step) and r is the outcome of its local random coin.

A *configuration* of the protocol assigns a local state to each process' automaton and a set of messages in transit (we call it the *message buffer*). The *initial configuration* C_{init} assigns the initial local state to each process and assumes that there are no messages in transit. A

step $s = (p, m, r)$ is *applicable to a configuration* C if m is $\perp$ or m is in the message buffer of C. The result of s applied to C is a new configuration $C.s$, where, based on the automaton of p, the local state of p is modified and finitely many messages are added to the message buffer.

A sequence of steps $E = s_1, s_2, \dots$ is applicable to a configuration C_0 if each s_i, $i = 1, 2, \dots$ is applicable to C_{i-1}, where, for all $i \geq 1$, $C_i = C_{i-1}.s_i$. We use $C_0.E$ to denote the resulting configuration; we also say that $C = C_0.E$ is an *extension of* C_0. By convention, C is a trivial extension of C.

An *execution* of the protocol is a sequence of steps E applicable to C_{init}. A finite execution E results in a *reachable* configuration C_E. Somewhat redundantly, we call a sequence of steps applicable to a reachable configuration C_E an *extension of* E. It immediately follows that steps of disjoint sets of processes commute:

▶ **Lemma 4.** *Let* C *be a reachable configuration, and* E *and* E' *be sequences of steps of disjoint sets of processes. If both* E *and* E' *are applicable to* C*, then* $C.E.E'$ *and* $C.E'.E$ *are identical reachable configurations.*

In an infinite execution, a process is *correct* if it executes infinitely many steps. We assume that every message m that is sent to a correct process p is eventually received, i.e., the execution will eventually contain a step $(p, m, -)$. As we assume that at most one process is allowed to fail by crashing, we only consider infinite executions in which at least $n - 1$ out of n processes are correct.

Without loss of generality, we assume that the implemented coin is binary: the correct processes either all output 0 or all output 1.

A configuration C is called *bivalent* if it has an extension $C.E_0$ in which some process outputs 0 and an extension $C.E_1$ in which some process outputs 1. Notice that any configuration preceding a bivalent configuration must be bivalent. Also, no process can produce a random-coin output in a bivalent configuration: otherwise, we get an execution in which two processes disagree on the output.

A protocol configuration that is not bivalent is called *univalent*: 0-valent if it has no extension in which 1 is decided or 1-valent otherwise.

▶ **Lemma 5.** *The initial configuration* C_{init} *is bivalent.*

Proof. The algorithm must output each of the two values with a positive probability. Thus, for each $v \in \{0, 1\}$, there exists an assignment of local coin outcomes and a message schedule that result in an execution with outcome v. ◀

▶ **Lemma 6.** *Let* C *be a reachable configuration, and* E *and* E' *be sequences of steps of a process* p *applicable to* C*. If* $C.E$ *and* $C.E'$ *are univalent, then they have the same valencies.*

Proof. The difference between $C.E$ and $C.E'$ consists in the local state of p and the message buffer. Since the algorithm is required to tolerate a single crash fault, there must exist a sequence of steps E'' that does not include any steps of p such that some process q outputs a value $v \in \{0, 1\}$ in $C.E''$. Moreover, as E'' contains no steps of p, E'' is also applicable to both $C.E$ and $C.E'$. But as the two configurations have opposite valences, and q decides the same value in $C.E.E''$ and $C.E'.E''$. Thus, if $C.E$ and $C.E'$ are univalent, then they must have the same valencies. ◀

Following the steps of [17], we show that the protocol must have a *critical* configuration D for which:

- there exist steps s_p and s_q of processes p and q such that both s_p and $s_q.s_p$ are applicable to D
- $D.s_p$ is 0-valent;
- $D.s_q.s_p$ is 1-valent.

▶ **Lemma 7.** *There must exist a critical configuration.*

Proof. Starting with the initial (bivalent by Lemma 5) configuration $C := C_{init}$, we pick process $p := p_1$ and check if there exists C', an extension of C, and s_p, a step of p applicable to C' in which the oldest message to p in the message buffer is consumed (if any), such that $C'.s_p$ is bivalent. If this is the case, we set C to $C'.s_p$, pick up the next process $p := p_2$. Again, if there exists C', an extension of C, and s_p, a step of p in which the oldest message to p in the message buffer is consumed, such that $C'.s_p$ is bivalent, then we set C to $C'.s_p$. We repeat the procedure, each time picking the next process (in the round-robin order, i.e., after p_n we go to p_1, etc.), as long as it is possible.

The first observation is that the procedure cannot be repeated indefinitely. Indeed, otherwise, we obtain an infinite sequence of steps in which every process takes infinitely many steps and receives every message sent to it (and which is, thus, an execution of the protocol) that goes through bivalent configurations only. Hence, this execution cannot produce an output – a contradiction with the Termination property.

Thus, there must exist a bivalent configuration C and a process p, such that for each C', an extension of C and each $s_p = (p, m, -)$, a step of p applicable to C', $C'.s_p$ is univalent, where m is the oldest message addressed to p in C.

Let $s_p = (p, m, r)$ be a step of p applicable to C. Without loss of generality, let $C.s_p$ be 0-valent.

As C is bivalent, it must have an extension $E = e_1, \dots, e_k$ such that $C.E$ is 1-valent.

Let ℓ be the largest index in $\{1, \dots, k\}$ such that either s_p is not applicable to $C_\ell = C.e_1, \dots, e_\ell$ or $C_\ell.s_p$ is 1-valent. Such an index exists, as $C.s_p$ is 0-valent $C.E$ is 1-valent and for any C', an extension of C, if s_p is applicable to C', then $C'.s_p$ is bivalent.

Suppose first that s_p is not applicable to C_ℓ. As s_p is applicable to every configuration $C_j = C.e_1, \dots, e_j$, $j = 1, \dots, \ell - 1$, e_ℓ must be be of the form (p, m, r'), i.e., e_ℓ must consume the message received in $s_p = (p, m, r)$. By our assumption $C_\ell = C_{\ell-1}.(p, m, r')$ is univalent. As C_ℓ has a 1-valent descendant C', it must be 1-valent too.

But, by Lemma 6, $C_{\ell-1}.s_p = C_{\ell-1}.(p, m, r)$ and $C_\ell = C_{\ell-1}.(p, m, r')$, must have the same valencies – a contradiction.

Thus, $C_\ell.s_p$ is 1-valent. Hence, we have a bivalent configuration $D = C_{\ell-1}$ and steps s_p and $s_q = e_\ell$ such that both s_p and $s_q.s_p$ are applicable to D. Moreover, $D.s_p$ is 0-valent and $D.s_q.s_p$ is 1-valent. Thus, we get a critical configuration. ◀

Finally, we establish a contradiction by showing that:

▶ **Lemma 8.** *No critical configuration may exist.*

Proof. By contradiction, let a critical configuration D exist, and let s_p and s_q be steps of p and q applicable to D such that $D.s_p$ is 0-valent and $D.s_q.s_p$ be 1-valent.

If s_q is a step of p, then Lemma 6 establishes a contradiction.

Otherwise, consider any infinite execution going through to $D.s_p$ in which all processes but q take infinitely many steps in this execution after $D.s_p$. By the Termination property, there must exist a finite sequence of steps E such that some process outputs a value $v \in \{0, 1\}$ in $D.s_p.E$. As $D.s_p$ is 0-valent, $v = 0$. Moreover, as E contains no steps of q and p has the same state in $D.s_p$ and $D.s_q.s_p$, E is also applicable to $D.s_q.s_p$. Thus, 0 is also decided in $D.s_q.s_p.E$ – a contradiction with the assumption that $D.s_q.s_p$ is 1-valent. ◀

Lemmata 7 and 8 imply:

▶ **Theorem 9.** *There does not exist a* 1*-resilient asynchronous random coin protocol.*

B Intersecting Random Subsets

A direct application of an approximate common coin is a problem that we call *intersecting random subsets*. In this problem the following information is globally known: a set S, where $|S| = s$; and two parameters m and k, where $1 \leq k \leq m \leq s$. Each correct process i chooses S_i, a random subset of S with cardinality m, such that $|\bigcap_{j\text{ – correct}} S_j| \geq m - k$. In other words, there are at least $m - k$ elements chosen at random that appear in all subsets.

The following variation of Gray Codes [23, 36] will be instrumental:

▶ **Definition 10.** *Code $C_{s,m}$ is a list of $\binom{s}{m}-1$ binary strings (called* code words*) satisfying the following conditions:*

- *for all i, $C_{s,m}[i]$ is a string of m ones and $s - m$ zeros;*
- *every string of m ones and $s - m$ zeros is present in $C_{s,m}$ exactly once;*
- *$\forall i \in \left\{1, \ldots, \binom{s}{m}-1\right\} : C_{s,m}[i]$ and $C_{s,m}[i-1]$ differ in two bits;*
- *$C_{s,m}[\binom{s}{m}-1]$ and $C_{s,m}[0]$ differ in two bits.*

The following recursive equation satisfies all requirements.

$$\forall s \geq 0 \text{ and } m \in [0..s] : C_{s,m} = \begin{cases} [\text{“ ”}] & \text{if } 0 = m = s \\ [\text{“}000\ldots000\text{”}] & \text{if } 0 = m < s \\ [\text{“}111\ldots111\text{”}] & \text{if } 0 < m = s \\ 0\|C_{s-1,m}, \text{reverse}(1\|C_{s,m-1}) & \text{otherwise} \end{cases}$$

In order to avoid computing all $\binom{s}{m}$ code words, when s and m are large, we can use the following recursive formula to efficiently (with $O(poly(s))$ operations) find a code word by its index:

$$\forall s \geq 0 \text{ and } m \in [0..s], i \in [0..\binom{s}{m}-1] :$$

$$C_{s,m}[i] = \begin{cases} \text{“ ”} & \text{if } 0 = m = s \\ \text{“}000\ldots000\text{”} & \text{if } 0 = m < s \\ \text{“}111\ldots111\text{”} & \text{if } 0 < m = s \\ 0\|C_{s-1,m}[i] & \text{if } 0 < m < s, i < \binom{s-1}{m} \\ 1\|C_{s-1,m-1}[\binom{s-1}{m-1} - \left(i - \binom{s-1}{m}\right) - 1] & \text{if } 0 < m < s, i \geq \binom{s-1}{m} \end{cases}$$

For example, here is $C_{5,2}$: [00011, 00110, 00101, 01100, 01010, 01001, 11000, 10100, 10010, 10001].

Intuitively, this code is composed of binary strings of length s and it can be read in the following manner: if the i-th position of the string is 1, then i-th element is selected, otherwise it is considered to be left outside. Moreover, this code has the property that consecutive numbers differ only by swapping exactly one position set to 1 with a position marked with a 0. Therefore all consecutive subsets have the same fixed size and include $m-1$ common elements and differ by only one. Hence, by generating an approximate common random coin over the domain $\left\{0..\binom{s}{m}-1\right\}$ with parameter $\epsilon \leq k \cdot \binom{s}{m}^{-1}$, processes can select subsets of size m differing by at most k elements.

This could be interesting for selecting a committee among n users in scenarios subject to a mobile Byzantine adversary, i.e. on systems where the set of processes who display malicious behaviour changes, provided the time to corrupt a majority of processes in any given committee is higher than an asynchronous round. Note that this solution provides an interesting alternative to committee elections in protocols such as Algorand [22]. It not only is completely asynchronous, but it also guarantees a fixed committee size and provides a way to control the intersection of quorums obtained by different users. Recall that in the case of Algorand, with non-zero probability, it might happen that quorums do not intersect at all.

Fragmented ARES: Dynamic Storage for Large Objects

Chryssis Georgiou
University of Cyprus, Nicosia, Cyprus

Nicolas Nicolaou
Algolysis Ltd, Limassol, Cyprus

Andria Trigeorgi
University of Cyprus, Nicosia, Cyprus

Abstract

Data availability is one of the most important features in distributed storage systems, made possible by data replication. Nowadays data are generated rapidly and developing efficient, scalable and reliable storage systems has become one of the major challenges for high performance computing. In this work, we develop and prove correct a dynamic, robust and strongly consistent distributed shared memory suitable for handling large objects (such as files) and utilizing erasure coding. We do so by integrating an Adaptive, Reconfigurable, Atomic memory framework, called ARES, with the COBFS framework, which relies on a block fragmentation technique to handle large objects. With the addition of ARES, we also enable the use of an erasure-coded algorithm to further split the data and to potentially improve storage efficiency at the replica servers and operation latency. Our development is complemented with an in-depth experimental evaluation on the Emulab and AWS EC2 testbeds, illustrating the benefits of our approach, as well as interesting tradeoffs.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms

Keywords and phrases Distributed storage, Large objects, Strong consistency, High access concurrency, Erasure code, Reconfiguration

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.25

Related Version *Full Version*: https://arxiv.org/abs/2201.13292

Supplementary Material *Dataset*: https://github.com/atrigeorgi/fragmentedARES-data.git [3]

1 Introduction

Motivation and prior work. Distributed Storage Systems (DSS) have gained momentum in recent years, following the demand for available, accessible, and survivable data storage [32, 34]. To preserve those properties in a harsh, asynchronous, fail prone environment (as a distributed system), data are replicated in multiple, often geographically separated devices, raising the challenge on how to preserve consistency between the replica copies.

For more than two decades, a series of works [11, 26, 19, 15, 18, 23] suggested solutions for building distributed shared memory emulations, allowing data to be shared concurrently offering basic memory elements, i.e. registers, with strict consistency guarantees. Linerazibility (atomicity) [24] is the most challenging, yet intuitive consistency guarantee that such solutions provide. The problem of keeping copies consistent becomes even more challenging when failed replica hosts (or servers) need to be replaced or new servers need to be added in the system. Since the data of a DSS should be accessible immediately, it is imperative that the service interruption during a failure or a repair should be as short as possible. The need to be able to modify the set of servers while ensuring service liveness yielded dynamic solutions and reconfiguration services. Examples of reconfigurable storage algorithms are RAMBO [22], DYNASTORE [7], SM-STORE [25], SPSNSTORE [17] and ARES [28].

© Chryssis Georgiou, Nicolas Nicolaou, and Andria Trigeorgi;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 25; pp. 25:1–25:24
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Currently, such reconfigurable emulations are limited to small-size, versionless, primitive objects (like registers), hindering the practicality of the solutions when dealing with larger, more common DSS objects (like files). *Coverability* [27] extends linearizability with the additional guarantee that object writes succeed when associating the written value with the "current" version of the object. In a different case, a write operation becomes a read operation and returns the latest version and the associated value of the object. This is essential, for example, for files. When updating the content of a file, one expects that the update is on the previous version of the file; linearizable registers do not impose such restriction, i.e., a write operation might change the value of the object arbitrarily, independently of the previously written value. A recent work by Anta et al. [8], introduced a modular solution, called CoBFS, which combines a suitable data fragmentation strategy, implemented as a Fragmentation module (FM), with a distributed shared memory module (DSMM), to efficiently handle and boost concurrency of large objects, while maintaining strong consistency guarantees (coverability and linearizability), and minimizing operation latencies. The fragmentation strategy enables two (or more) concurrent write operations on different fragments of the object to both take effect, without violating the consistency of the object as a whole. These solutions, as well as the one proposed in [16], were designed for the static environment (fixed set of servers). *In this work we study whether it is plausible to bring coverability and fragmentation to dynamic environments, and how challenging such adaptation would be.*

Contributions. This work is the first to consider dynamic (reconfigurable) Distributed Shared Memory (DSM) tailored for versioned (coverable) and large (fragmentable) objects. At the same time, we aim to introduce solutions that maximize the concurrency of operations on the shared object while trading consistency on the whole object. In particular, we propose a dynamic DSM that: (*i*) supports versioned objects, (*ii*) is suitable for large objects (such as files), and (*iii*) is storage-efficient. To achieve this, we integrate the dynamic DSM algorithm Ares [28] with the DSMM module in CoBFS. Ares is the first algorithm that enables erasure coded based dynamic DSM yielding benefits on the storage efficiency at the replica hosts. To support versioning we extend Ares to implement coverable objects, while high access concurrency is preserved by introducing support for fragmented objects. Ultimately, we aim to make a leap towards dynamic DSS that will be attractive for practical applications (like highly concurrent and strongly consistent file sharing).

In summary, our contributions are the following:

- We propose and prove the correctness of the coverable version of Ares, CoAres, the first Fault-tolerant, Reconfigurable, Erasure coded, Atomic Memory, to support versioned objects (Section 4).
- We adopt the idea of fragmentation as presented in CoBFS [8], to obtain CoAresF, which enables CoAres to handle *large* shared data objects and increased data access concurrency (Section 5). The correctness of CoAresF is rigorously proven.
- To reduce the operational latency of the read/write operations in the DSMM layer, we apply and prove correct an optimization in the implementation of the erasure coded *data-access primitives* (DAP) used by the Ares framework (which includes CoAres and CoAresF). This optimization has its own interest, as it could be applicable beyond the Ares framework, i.e., by other erasure coded algorithms relying on tag-ordered DAPs (Section 6).
- We have performed an in-depth experimental evaluation of our approach over both Emulab and Amazon Web Services (AWS) EC2 (Section 7). Our experiments compare various versions of our implementation, i.e., with and without the fragmentation technique or with and without Erasure Code or with and without reconfiguration, illustrating tradeoffs and synergies.

We note that although the extension of Ares and its integration with CoBFS might appear conceptually simple, handling reconfiguration was quite subtle, and proving the correctness of the integration was non-trivial. Appendix A provides a table comparing our work with other distributed storage algorithms and systems.

2 Model and Definitions

In this section we present the system setting and define necessary terms we use in the rest of the manuscript. As mentioned, our main goal is to implement a reconfigurable strongly consistent shared memory that supports large shared objects and favors high access concurrency. We assume read/write (R/W) shared objects that support two operations: (i) a *read operation* that returns the value of the object, and (ii) a *write operation* that modifies the value of the object.

Executions and histories An *execution* ξ of a distributed algorithm A is an alternating sequence of *states* and *actions* of A reflecting the evolution in real time of the execution. A history H_ξ is the subsequence of the actions in ξ. A history H_ξ is *sequential* if it starts with an invocation action and each invocation is immediately followed by its matching response; otherwise, H_ξ is *concurrent*. Finally, H_ξ is *complete* if every invocation in H_ξ has a matching response in H_ξ, i.e., each operation in ξ is complete. An operation π_1 precedes an operation π_2 (or π_2 succeeds π_1), denoted by $\pi_1 \rightarrow \pi_2$, in H_ξ, if the response action of π_1 precedes the invocation action of π_2 in H_ξ. Two operations are concurrent if none precedes the other.

Clients and servers. We consider a system composed of four distinct sets of crash-prone, asynchronous processes: a set $\mathcal{W}$ of writers, a set $\mathcal{R}$ of readers, a set $\mathcal{G}$ of reconfiguration clients, and a set $\mathcal{S}$ of servers. Let $\mathcal{I} = \mathcal{W} \cup \mathcal{R} \cup \mathcal{G}$ be the set of clients. Servers host data elements (replicas or encoded data fragments). Each writer is allowed to modify the value of a shared object, and each reader is allowed to obtain the value of that object. Reconfiguration clients attempt to introduce new configuration of servers to the system in order to mask transient errors and to ensure the longevity of the service. (In our implementations, a client can perform any operation.)

Configurations. A configuration, $c \in \mathcal{C}$, consists of: (i) $c.Servers \subseteq \mathcal{S}$: a set of server identifiers; (ii) $c.Quorums$: the set of quorums on $c.Servers$, s.t. $\forall Q_1, Q_2 \in c.Quorums, Q_1, Q_2 \subseteq c.Servers$ and $Q_1 \cap Q_2 \neq \emptyset$; (iii) $DAP(c)$: the set of data access primitives (operations at level lower than reads or writes) that clients in $\mathcal{I}$ may invoke on $c.Servers$ (cf. Section 3); (iv) $c.Con$: a consensus instance with the values from $\mathcal{C}$, implemented as a service on top of the servers in $c.Servers$; and (v) the pair $(c.tag, c.val)$: the maximum tag-value pair that clients in $\mathcal{I}$ have. A tag consists of a timestamp ts (sequence number) and a writer id; the timestamp is used for ordering the operations, and the writer id is used to break symmetry (when two writers attempt to write concurrently using the same timestamp) [22]. We refer to a server $s \in c.Servers$ as a *member* of configuration c.

Fragmented objects. As defined in [8], a *fragmented object* is a totally ordered sequence of *block objects*. Let $\mathcal{F}$ denote the set of fragmented objects, and $\mathcal{B}$ the set of block objects. A block $b \in \mathcal{B}$ is a concurrent R/W object with a unique id and is associated with two structures, *val* and *ver*: $val(b)$ is composed of a *pointer* that points to the next block in the sequence, and the *data* contained in the block; $ver(b) = \langle wid, bseq \rangle$, where $wid \in \mathcal{I}$ is the id

of a writer and $bseq \in \mathbb{N}$ is a sequence number (initially 0). A *fragmented object* $f \in \mathcal{F}$ is a *sequence* of blocks from $\mathcal{B}$, with a value $val(f) = \langle b_0, b_1, b_2, \ldots \rangle$, where each $b_i \in \mathcal{B}$. Initially, a fragmented object contains an empty block, i.e., $val(f) = \langle b_0 \rangle$ with $val(b_0) = \varepsilon$; we refer to it as the *genesis* block.

Coverability and Fragmented Coverability. Our goal is to implement *fragmented linearizable coverable* objects. Linearizability [24] provides the illusion that a concurent object is accessed sequentially when in reality is accessed concurrently by multiple processes. *Coverability* is defined over a *totally ordered* set of *versions* and introduces the notion of *versioned objects*. According to [27], a *versioned object* is a type of R/W object where each value written is assigned with a version. *A coverable object* is a versioned object satisfying the properties *consolidation*, *continuity* and *evolution*.

Intuitively, *consolidation* specifies that write operations may revise the object with a version larger than any version modified by a preceding write operation, and may lead to a version newer than any version introduced by a preceding write operation. *Continuity* requires that a write operation may revise a version that was introduced by a preceding write operation, according to the given total order. Finally, *evolution* limits the relative increment on the version of an object that can be introduced by any operation. Their formal definitions are given in Section 4.

In [27], the notion of a *successful* and *unsuccessful* write was introduced. A successful write is denoted as $cvr\text{-}\omega(ver)[ver', chg]_p$, which updates the object from version ver to ver' (along with the associated values), whereas an unsuccessful write is denoted as $cvr\text{-}\omega(ver)[ver', unchg]_p$ (i.e., it becomes a read). Note that in [27], *ver*s were implemented as *tag*s.

Fragmented linearizable coverability [8] guarantees that concurrent write operations on different blocks would *all* prevail (as long as each write is tagged with the latest version of each block), whereas only one write operation on the same block eventually prevails (all other concurrent writes operations on the same block would become read operations).Thus, a fragmented object implementation satisfying this property may lead to higher access concurrency [8].

3 ARES: A Framework for Dynamic Storage

ARES [28] is a modular framework, designed to implement dynamic, reconfigurable, fault-tolerant, read/write distributed linearizable (atomic) shared memory objects.

Similar to traditional implementations, ARES uses $\langle tag, value \rangle$ pairs to order the operations on a shared object. In contrast to existing solutions, ARES does not define the exact methodology to access the object replicas. Rather, it relies on three, so called, *data access primitives* (DAPs): (*i*) the get-tag, which returns the tag of an object, (*ii*) the get-data, which returns a $\langle tag, value \rangle$ pair, and (*iii*) the put-data($\langle \tau, v \rangle$), which accepts a $\langle tag, value \rangle$ as an argument.

As seen in [28], these DAPs may be used to express the data access strategy (i.e., how they retrieve and update the object data) of different shared memory algorithms (e.g., [10]). Using the DAPs, ARES achieves a modular design, agnostic of the data access strategies, and enables the use of different DAP implementation per configuration (something impossible for other solutions). For the DAPs to be useful, they need to satisfy a property, referred in [28] as **Property 1**, which involves two conditions: **(C1)** if a put-data($\langle \tau, v \rangle$ precedes a get-data (or get-tag) operation, then the latter operation returns a value associated with a tag $\tau' \geq \tau$, and **(C2)** if a get-data returns $\langle \tau', v' \rangle$ then there exists put-data($\langle \tau', v' \rangle$ that precedes or is concurrent to the get-data operation. A formal definition appears in [28].

DAP Implementations. To demonstrate the flexibility that DAPs provide, the authors in [28], expressed two different atomic shared R/W algorithms in terms of DAPs. These are the DAPs for the well celebrated ABD [10] algorithm, and the DAPs for an erasure coded based approach presented for the first time in [28]. In the rest of the manuscript we refer to the two DAP implementations as ABD-DAP and EC-DAP. An $[n, k]$-MDS erasure coding algorithm (e.g., Reed-Solomon [31]) encodes k object fragments into n coded elements, which consist of the k encoded data fragments and m encoded parity fragments. The n coded fragments are distributed among a set of n different servers. Any k of the n coded fragments can then be used to reconstruct the initial object value. As servers maintain a fragment instead of the whole object value, EC based approaches claim significant storage benefits. By utilizing the EC-DAP, Ares became *the first* erasure coded dynamic algorithm to implement an atomic R/W object.

We now provide a high-level description of the two main functionalities supported by Ares: (i) the reconfiguration of the servers, and (ii) the read/write operations on the shared object.

Reconfiguration. Reconfiguration is the process of changing the set of servers. A configuration sequence $cseq$ in Ares is defined as a sequence of pairs $\langle c, status \rangle$ where $c \in \mathcal{C}$, and $status \in \{P, F\}$ (P stands for pending and F for finalized). Configuration sequences are constructed and stored in clients, while each server in a configuration c only maintains the configuration that follows c in a local variable $nextC \in \mathcal{C} \cup \{\perp\} \times \{P, F\}$.

To perform a reconfiguration operation recon(c), a client r follows 4 steps. At first, r executes a sequence traversal to discover the latest configuration sequence $cseq$. Then it attempts to add $\langle c, P \rangle$ at the end of $cseq$ by proposing c to a consensus mechanism. The outcome of the consensus may be a configuration c' (possibly different than c) proposed by some reconfiguration client. Then the client determines the maximum tag-value pair of the object, say $\langle \tau, v \rangle$ by executing get-data operation and transfers the pair to c' by performing put-data($\langle \tau, v \rangle$) on c'. Once the update of the value is complete, the client finalizes the proposed configuration by setting $nextC = \langle c', F \rangle$ in a quorum of servers of the last configuration in $cseq$ (or c_0 if no other configuration exists). As shown in [28], this reconfiguration procedure guarantees that configuration sequences obtained by any two clients $cseq_p$ and $cseq_q$, then either $cseq_p$ is a prefix of $cseq_q$, or vice versa.

Read/Write operations. A write (or read) operation π by a client p is executed by performing the following actions: (i) π invokes a read-config action to obtain the latest configuration sequence $cseq$, (ii) π invokes a get-tag (if a write) or get-data (if a read) in each configuration, starting from the last finalized to the last configuration in $cseq$, and discovers the maximum τ or $\langle \tau, v \rangle$ pair respectively, and (iii) repeatedly invokes put-data($\langle \tau', v' \rangle$), where $\langle \tau', v' \rangle = \langle \tau + 1, v' \rangle$ if π is a write and $\langle \tau', v' \rangle = \langle \tau, v \rangle$ if π is a read in the last configuration in $cseq$, and read-config to discover any new configuration, until no additional configuration is observed.

4 CoAres: Coverable Ares

In this section we present and analyze the coverable extension of Ares, which we refer to as CoAres.

Description. Below we describe the modification that need to occur on Ares in order to support coverability. The reconfiguration protocol and the DAP implementations remain the same as they are not affected by the application of coverability. The changes occur in the specification of read/write operations, which we detail below.

Read/Write operations. Algorithm 1 specifies the read and write protocols of CoAres. The blue text annotates the changes when compared to the original Ares read/write protocols. The local variable $flag \in \{chg, unchg\}$, maintained by the write clients, is set to chg when the write operation is successful and to $unchg$ otherwise; initially it is set to $unchg$. The state variable $version$ is used by the client to maintain the tag of the coverable object. At first, in both cvr-read and cvr-write operations, the read/write client issues a read-config action to obtain the latest introduced configuration; cf. line Alg. 1:14 (resp. line Alg. 1:43).

In the case of cvr-write, the writer w_i finds the last finalized entry in $cseq$, say μ, and performs a $cseq[j].conf$.get-data() action, for $\mu \leq j \leq |cseq|$ (lines Alg. 1:15–18). Thus, w_i retrieves all the $\langle \tau, v \rangle$ pairs from the last finalized configuration and all the pending ones. Note that in cvr-write, get-data is used in the first phase instead of a get-tag, as the coverable version needs both the highest tag and value and not only the tag, as in the original write protocol. Then, the writer computes the maximum $\langle \tau, v \rangle$ pair among all the returned replies. Lines Alg. 1:19 - 1:24 depict the main difference between the coverable cvr-write and the original one: if the maximum τ is equal to the state variable $version$, meaning that the writer w_i has the latest version of the object, it proceeds to update the state of the object ($\langle \tau, v \rangle$) by increasing τ and assigning $\langle \tau, v \rangle$ to $\langle \langle \tau.ts + 1, \omega_i \rangle, val \rangle$, where val is the value it wishes to write (lines Alg. 1:20–21). Otherwise, the state of the object does not change and the writer keeps the maximum $\langle \tau, v \rangle$ pair found in the first phase (i.e., the write has become a read). No matter whether the state changed or not, the writer updates its $version$ with the value τ (line Alg. 1:24).

Algorithm 1 Write and Read protocols for CoAres.

```
    CVR-Write Operation:
2:    at each writer w_i
      State Variables:
4:    cseq[] s.t. cseq[j] ∈ C × {F, P}
      version ∈ ℕ⁺ × W ∪ {⊥} initially ⟨0, ⊥⟩
6:    Local Variables:
      μ ∈ ℕ⁺ initially 0, ν ∈ ℕ⁺ initially 0
8:    τ ∈ ℕ⁺ × W initially ⟨0, w_i⟩
      v ∈ V initially ⊥
10:   flag ∈ {chg, unchg} initially unchg
      Initialization:
12:   cseq[0] = ⟨c_0, F⟩

      operation cvr-write(val), val ∈ V
14:     cseq ← read-config(cseq)
        μ ← max({i : cseq[i].status = F})
16:     ν ← |cseq|
        for i = μ : ν do
18:       ⟨τ, v⟩ ← max(cseq[i].cfg.get-data(), ⟨τ, v⟩)
        if version = τ then
20:       flag ← chg
          ⟨τ, v⟩ ← ⟨⟨τ.ts + 1, ω_i⟩, val⟩
22:     else
          flag ← unchg
24:     version ← τ
        done ← false
26:     while not done do
          cseq[ν].cfg.put-data(⟨τ, v⟩)
28:       cseq ← read-config(cseq)
          if |cseq| = ν then
30:         done ← true
          else
32:         ν ← |cseq|
        end while
34:     return ⟨τ, v⟩, flag
      end operation
36: CVR-Read Operation:
      at each reader r_i
38:   State Variables:
      cseq[] s.t. cseq[j] ∈ C × {F, P}
40:   Initialization:
      cseq[0] = ⟨c_0, F⟩

42:   operation cvr-read( )
        cseq ← read-config(cseq)
44:     μ ← max({j : cseq[j].status = F})
        ν ← |cseq|
46:     for i = μ : ν do
          ⟨τ, v⟩ ← max(cseq[i].cfg.get-data(), ⟨τ, v⟩)
48:     done ← false
        while not done do
50:       cseq[ν].cfg.put-data(⟨τ, v⟩)
          cseq ← read-config(cseq)
52:       if |cseq| = ν then
            done ← true
54:       else
            ν ← |cseq|
56:     end while
        return ⟨τ, v⟩
58:   end operation
```

In the case of cvr-read, the first phase is the same as the original, that is, it discovers the *maximum tag-value* pair among the received replies (lines Alg. 1:46–47). The propagation of $\langle \tau, v\rangle$ in both cvr-write (lines Alg. 1:26–33) and cvr-read (lines Alg. 1:49–56)) remains the same. Finally, the cvr-write operation returns $\langle \tau, v\rangle$ and the $flag$, whereas the cvr-read operation only returns ($\langle \tau, v\rangle$).

Correctness of COARES. CoAres is correct if it satisfies *liveness* (termination) and *safety* (i.e., linearizable coverability). Termination holds since read, update and reconfig operations on the CoAres always complete given that the DAP completes. As shown in [28], Ares implements a linearizable object given that the DAP used satisfy Property 1. Given that CoAres uses the same reconfiguration and read operations, while the write operation might get converted to a read operation, then linearizability is not affected and can be shown that it holds in a similar way as in [28].

The validity and coverability properties, defined formally below as Definitions 1 and 2, remain to be examined. In CoAres, we use tags to denote the version of the register. Given that the $DAP(c)$ used in any configuration $c \in \mathcal{C}$ satisfies Property 1, we will show that any execution ξ of CoAres satisfies the properties of Definitions 1 and 2.

Proof challenges: The main challenge is to show that CoAres satisfies the coverability properties despite *any reconfiguration* in the system. In particular, we would like to ensure: (i) new values are not overwritten, i.e., if a write is successfully completed then no subsequent write successfully writes a value associated with an older version in any active configuration, (ii) versions are unique, and (iii) eventually a single version path prevails.

Definitions and proofs: In the lemmas that follow, we refer to a successful write operation as one that is not converted to a read operation. We say that a write operation *revises* a version ver of the versioned object to a version ver', or *produces* ver', in an execution ξ, if $cvr\text{-}\omega(ver)[ver']_{p_i}$ completes in H_ξ. Let the set of *successful write* operations on a history H_ξ be defined as $\mathcal{W}_{\xi,succ} = \{\pi : \pi = cvr\text{-}\omega(ver)[ver']_{p_i} \text{ completes in } H_\xi\}$. The set of the object's versions produced by writes operations in the history H_ξ is defined by $Versions_\xi = \{ver_i : cvr\text{-}\omega(ver)[ver_i]_{p_i} \in \mathcal{W}_{\xi,succ}\} \cup \{ver_0\}$, where ver_0 is the initial version of the object. Observe that the elements in $Versions_\xi$ are totally ordered. Now we present the *validity* property which defines explicitly the set of executions that are considered to be valid executions.

▶ **Definition 1** (Validity [27]). *An execution ξ (resp. its history H_ξ) is a* valid execution *(resp. history) on a versioned object, for any $p_i, p_j \in \mathcal{I}$:*

1. $\forall cvr\text{-}\omega(ver)[ver']_{p_i} \in \mathcal{W}_{\xi,succ}, ver < ver'$,
2. *for any operations $cvr\text{-}\omega(*)[ver']_{p_i}$ and $cvr\text{-}\omega(*)[ver'']_{p_j}$ in $\mathcal{W}_{\xi,succ}$, $ver' \neq ver''$, and*
3. *for each $ver_k \in Versions_\xi$ there is a sequence of versions $ver_0, ver_1, \ldots, ver_k$, such that $cvr\text{-}\omega(ver_i)[ver_{i+1}] \in \mathcal{W}_{\xi,succ}$, for $0 \leq i < k$.*

▶ **Definition 2** (Coverability [27]). *A valid execution ξ is* ***coverable*** *with respect to a total order $<_\xi$ on operations in $\mathcal{W}_{\xi,succ}$ if:*

1. **(Consolidation)** *If $\pi_1 = cvr\text{-}\omega(*)[ver_i], \pi_2 = cvr\text{-}\omega(ver_j)[*] \in \mathcal{W}_{\xi,succ}$, and $\pi_1 \rightarrow_{H_\xi} \pi_2$ in H_ξ, then $ver_i \leq ver_j$ and $\pi_1 <_\xi \pi_2$.*
2. **(Continuity)** *if $\pi_2 = cvr\text{-}\omega(ver)[ver_i] \in \mathcal{W}_{\xi,succ}$, then there exists $\pi_1 \in \mathcal{W}_{\xi,succ}$ s.t. $\pi_1 = cvr\text{-}\omega(*)[ver]$ and $\pi_1 <_\xi \pi_2$, or $ver = ver_0$.*
3. **(Evolution)** *let $ver, ver', ver'' \in Versions_\xi$. If there are sequences of versions $ver'_1, ver'_2, \ldots, ver'_k$ and $ver''_1, ver''_2, \ldots, ver''_\ell$, where $ver = ver'_1 = ver''_1$, $ver'_k = ver'$, and $ver''_\ell = ver''$ such that $cvr\text{-}\omega(ver'_i)[ver'_{i+1}] \in \mathcal{W}_{\xi,succ}$, for $1 \leq i < k$, and $cvr\text{-}\omega(ver''_i)[ver''_{i+1}] \in \mathcal{W}_{\xi,succ}$, for $1 \leq i < \ell$, and $k < \ell$, then $ver' < ver''$.*

We proceed with formal statements and proofs. Lemmas 3 to 5 help us show that CoAres satisfies *Validity*.

▶ **Lemma 3** (Version Increment). *In any execution ξ of* CoAres*, if ω is a successful write operation, and ver the maximum version it discovered during the* get-data *operation, then ω propagates a version $ver' > ver$.*

Proof. This lemma follows from the fact that CoAres uses a condition before the propagation phase in line Alg. 1:19. The writer checks if the maximum tag retrieved from the get-data action is equal to the local *version*. If that holds, then the writer generates a new version larger than its local version by incrementing the tag found. ◀

▶ **Lemma 4** (Version Uniqueness). *In any execution ξ of* CoAres*, if two write operations ω_1 and ω_2, write values associated with versions ver_1 and ver_2 respectively, then $ver_1 \neq ver_2$.*

Proof. A tag is composed of an integer timestamp ts and the id of a process wid. Let w_1 be the id of the writer that invoked ω_1 and w_2 the id of the writer that invoked ω_2. To show whether the versions generated by the two write operations are not equal we need to examine two cases: (a) both ω_1 and ω_2 are invoked by the same writer, i.e. $w_1 = w_2$, and (b) ω_1 and ω_2 are invoked by two different writers, i.e. $w_1 \neq w_2$.

Case a: In this case, the uniqueness of the versions is achieved due to the well-formedness assumption and the $C1$ term in Property 1. By well-formdness, writer w_1 can only invoke one operation at a time. Thus, the last put-data$(ver_1, *)$ of ω_1 completes before the first get-data of ω_2.

If both operations are invoked and completed in the same configuration c then by $C1$, the version ver' returned by c.get-data, is $ver' \geq ver_1$. Since the version is incremented in ω_2 then $ver_2 = ver' + 1 > ver_1$, and hence $ver_1 \neq ver_2$ as desired.

It remains to examine the case where the put-data was invoked in a configuration c and the get-data in a configuration c'. Since by well-formedness $\omega_1 \to \omega_2$, then by the sequence prefix guaranteed by the reconfiguration protocol of Ares (second property) the $cseq_1$ obtained during the read-config action in ω_1 is a prefix of the $cseq_2$ obtained during the same action in ω_2. Notice that c' is the last finalized configuration in $cseq_2$ as this is the configuration where the first get-data action of ω_2 is invoked. If c' precedes c in $cseq_2$ then by CoAres the write operation ω_2 will invoke a get-data operation in c as well and with the same reasoning as before will generate a $ver_2 \neq ver_1$. If now c precedes c' in $cseq_2$, then it must be the case that a reconfiguration operation r has been invoked concurrently or after ω_2 and added c'. By Ares [28], r, invoked a put-data(ver') in c' before finalizing c' with $ver' \geq ver_1$. So when ω_2 invokes get-data in c' by $C1$ will obtain a version $ver'' \geq ver' \geq ver_1$. Hence $ver_2 > ver''$ and thus $ver_2 \neq ver_1$ as needed.

Case b: When $w_1 \neq w_2$ then ω_1 generates a version $ver_1 = \{ts_1, w_1\}$ and ω_2 generates some version $ver_2 = \{ts_2, w_2\}$. Even if $ts_1 = ts_2$ the two version differ on the unique id of the writers and hence $ver_1 \neq ver_2$. This completes the case and the proof. ◀

▶ **Lemma 5.** *Each version we reach in an execution is derived (through a chain of operations) from the initial version of the register ver_0.*

Proof. Every tag is generated by extending the tag retrieved by a get-data operation starting from the initial tag (lines Alg. 1:20–21). In turn, each get-data operation returns a tag written by a put-data operation or the initial tag (as per $C2$ in Property 1). Then, applying a simple induction, we may show that there is a sequence of tags leading from the initial tag to the tag used by the write operation. ◀

From this point onward we fix ξ to be a valid execution and H_ξ to be its valid history. We now show coverability (Definition 2).

▶ **Lemma 6.** *In any execution ξ of* CoAres*, all properties of Definition 2 are satisfied.*

Proof. For *consolidation* we need to show that for two write operations $\omega_1 = cvr\text{-}\omega(*)[\tau_1, chg]$ and $\omega_2 = cvr\text{-}\omega(\tau_2)[*, chg]$, if $\omega_1 \to_\xi \omega_2$ then $\tau_1 \leq \tau_2$. According to $C1$ of Property 1, since the get-data of ω_2 appears after the put-data of ω_1, the get-data of ω_2 returns a tag higher than the one written by ω_1.

Continuity is preserved as a write operation first invokes a get-data action for the latest tag before proceeding to put-data to write a new value. According to $C2$ of Property 1, the get-data action returns a tag already written by a put-data or the initial tag of the register.

To show that *evolution* is preserved, we take into account that the version of a register is given by its tag, where tags are compared lexicographically. A successful write $\pi_1 = cvr\text{-}\omega(\tau)[\tau']$ generates a new tag τ' from τ such that $\tau'.ts = \tau.ts + 1$ (line Alg. 1:21). Consider sequences of tags $\tau_1, \tau_2, \ldots, \tau_k$ and $\tau'_1, \tau'_2, \ldots, \tau'_\ell$ such that $\tau_1 = \tau'_1$. Assume that $cvr\text{-}\omega(\tau_i)[\tau_{i+1}]$, for $1 \leq i < k$, and $cvr\text{-}\omega(\tau'_i)[\tau'_{i+1}]$, for $1 \leq i < \ell$, are successful writes. If $\tau_1.ts = \tau'_1.ts = z$, then $\tau_k.ts = z + k$ and $\tau'_\ell.ts = z + \ell$, and if $k < \ell$ then $\tau_k < \tau'_\ell$. ◀

Lemmas 3 to 6 show that CoAres satisfies validity (Def. 1) and coverability (Def. 2):

▶ **Theorem 7.** CoAres *implements a linearizable coverable object, given that the DAPs implemented in any configuration c satisfy Property 1.*

5 CoAresF: Integrate CoAres with a Fragmentation approach

The work in [8] developed a distributed storage framework, called CoBFS, which utilizes coverable fragmented objects. In this section we describe how CoAres can be integrated with CoBFS to obtain what we call CoAresF, thus yielding a dynamic distributed memory suitable for large objects. Furthermore, this enables to combine the fragmentation approach of CoBFS with a second level of striping when EC-DAP is used, making storage efficient at the servers. A particular challenge of this integration is how the fragmentation approach should invoke reconfiguration operations, since CoBFS in [8] considered only static (non-reconfigurable) systems. The **main challenge** of CoAresF, however, was to prove that the blocks' sequence of a fragmented object remains connected, despite the existence of concurrent read/write and reconfiguration operations.

Overview of COBFS. The architecture of CoBFS is shown in Fig. 1 and it is composed of two main modules: (*i*) a Fragmentation Module (FM), and (*ii*) a Distributed Shared Memory Module (DSMM). In summary, the FM implements the fragmented object, which is a totally ordered sequence of blocks (where a block is a R/W object with limited value size; cf. Section 2), while the DSMM implements an interface to a shared memory service that allows operations on individual block objects. To this respect, CoBFS is flexible enough to utilize *any* underlying distributed shared memory implementation.

CoBFS mainly supports two operations, update and read, described next.

Update Operation. The update operation spans both modules, FM and DSMM. The FM uses a **Block Identification (BI) module**, which draws ideas from the RSYNC (Remote Sync) algorithm [33]. The BI includes three main modules, the *Block Division*, the *Block Matching* and *Block Updates*.

Figure 1 Basic architecture of CoBFS [8].

1. *Block Division:* This module splits a given fragmented object f into blocks. (This can be done, for example, in files, by using rolling hashing, such as *rabin fingerprints* [30], as we've done in our implementation.)
2. *Block Matching:* This module is used to find the differences between the new and the old blocks, yielding four *statuses*: (i) equality, (ii) modified, (iii) inserted, (iv) deleted. (As in our implementation, this can be done by using a string matching algorithm [13].)
3. *Block Updates:* Based on the retrieved statuses, the blocks of the fragmented object are then written on the DSM using the DSMM as an external service. In the case of equality, no operation is performed. In the case of modification, an *update* operation attempts to write the modified block. If new blocks are inserted after an existing block b, the *update* operation first writes the new blocks and then writes b so that the list of blocks remains connected. Delete is treated as a modification that sets an empty value to a block.

Read Operation. When the system receives a read request from a client, the FM issues, to the DSMM, a series of read operations on the fragmented object's blocks, starting from the genesis block and proceeding to the last block by following the next block ids. As blocks are retrieved, they are assembled as a fragmented object.

Integration of COARES in COBFS. Integration with the CoBFS is achieved by using CoAres as the external DSMM service. To accommodate the dynamic nature of CoAres, we need to introduce the reconfiguration operation in CoAresF as shown next.

Reconfig Operation. The specification of reconfig on the DSMM is given in Alg. 2, while the specification of reconfig on a fragmented object is given in Alg. 3.

When the system receives a reconfig request from a client, the FM issues a series of reconfig operations on the fragmented object's blocks, starting from the genesis block and proceeding to the last block by following the next block ids (Alg. 3). The $reconfig$ operation executes the $block\ reconfig$ operations on the shared memory (Alg. 2) using dsmm-reconfig operations.

Algorithm 2 DSMM: Reconfig operation on block b at client p.

```
1: function dsmm-reconfig(c)_{b,p}
2:     b.reconfig(c)
3: end function
```

Algorithm 3 FM: Reconfig operation on fragmented object f at client p.

```
1: State Variables:
2: L_f a sequence of blocks, initially ⟨b_0⟩;

3: function fm-reconfig(c)_{f,p}
4:    b ← val(b_0).ptr
5:    L_f ← ⟨b_0⟩
6:    while b not NULL do
7:       dsmm-reconfig(c)_{b,p}
8:       b ← val(b).ptr
9:    end while
10: end function
```

Correctness of CoAresF. When a reconfig(c) operation is invoked in CoAres, a reconfiguration client requests to change the configuration of the servers hosting the single R/W object. By design, each instance of CoAres handles a single R/W object. In the case of a fragmented object f, each block composing f is handled as a separate atomic object, and thus assigned to a different Ares instance. Therefore, the **main challenge** of CoAresF is to ensure that the sequence composing f remains connected and composed of the most recent blocks, despite concurrent read/write and reconfig operations. Note that each individual block may exist in different configurations and be accessed by different DAPs.

In the remainder we show that *fragmented coverability* (see Section 2) cannot be violated. Before we prove any lemmas, we first state a claim that follows directly from the algorithm.

▷ Claim 8. For any block $b \neq b_0$, where b_0 the genesis block, created by an update operation, it is initialized with a configuration sequence $cseq_b = cseq_0$, where $cseq_0$ is the initial configuration.

Notice that we assume that a single quorum remains correct in $cseq_0$ at any point in the execution. This may change in practical settings by having an external service to maintain and distribute the latest $cseq$ that will be used in a created block.

We begin with a lemma that states that for any block in the sequence obtained by a read operation, there is a successful update operation that wrote this block. Its proof follows the proof of Lemma 4 presented in [9].

▶ **Lemma 9.** *In any execution ξ of* CoAresF, *if ρ is a read operation on f that returns a sequence $\mathcal{L}$, then for any block $b \in \mathcal{L}$, there exists a successful update operation on f that either precedes or is concurrent to ρ.*

In the following lemma we show that a reconfiguration moves a version of the object larger than any version written by a preceding write operation to the installed configuration.

▶ **Lemma 10.** *Suppose that ρ is a* dsmm-reconfig$(c_2)_{b,*}$ *operation and ω a successful* cvr-write$(v)_{b,*}$ *operation that changes the version of b to ver, s.t. $\omega \to \rho$ in an execution ξ of* CoAresF. *Then ρ invokes c_2.*put-data$(\langle ver', * \rangle)$ *in c_2, s.t. $ver' \geq ver$.*

Proof. Let $cseq_\omega$ be the last configuration sequence returned by the read-config action at ω (Alg. 1:28), and $cseq_\rho$ the configuration sequence returned by the first read-config action at ρ (see Alg. 2:8 in [28]). By the prefix property of the reconfiguration protocol, $cseq_\omega$ will be a prefix of $cseq_\rho$.

Let c_ℓ the last configuration in $cseq_\omega$, and c_1 the last finalized configuration in $cseq_\rho$. There are two cases to examine: (i) c_1 precedes c_ℓ in $cseq_\rho$, and (ii) c_1 appears after c_ℓ in $cseq_\rho$. If (i) is the case then during the update-config action, ρ will perform a c_ℓ.get-data() action. By $C1$ in Property 1, the c_ℓ.get-data() will return a version $ver'' \geq ver$. Since the ρ function will execute c_2.put-data$(\langle ver', * \rangle)$, s.t. ver' is the max discovered version, then $ver' \geq ver'' \geq ver$.

In case (ii) it follows that the reconfiguration operation that proposed c_1 has finalized the configuration. So either that reconfiguration operation moved a version ver'' of b s.t. $ver'' \geq ver$ in the same way as described in case (i) in c_1, or the write operation would observe c_1 during a read-config action. In the latter case c_1 will appear in $cseq_\omega$ and ω will invoke a c_ℓ.put-data($\langle ver, * \rangle$) s.t. either $c_\ell = c_1$ or c_ℓ a configuration that appears after c_1 in $cseq_\omega$. Since c_1 is the last finalized configuration in $cseq_\rho$, then in any of the cases described ρ will invoke a c_ℓ.get-data(). Thus, it will discover and put in c_2 a version $ver' \geq ver$ completing our proof. ◀

Next we need to show that any sequence returned by any read operation is connected, despite any reconfiguration operations that may be executed concurrently. *This corresponds to the most challenging part of the integration.*

▶ **Lemma 11.** *In any execution ξ of* CoAresF*, if ρ is a read operation on f that returns a sequence of blocks $\mathcal{L} = \{b_0, b_1, \ldots, b_n\}$, then it must be the case that (i) $b_0.ptr = b_1$, (ii) $b_i.ptr = b_{i+1}$, for $i \in [1, n-1]$, and (iii) $b_n.ptr = \bot$.*

Proof. Assume by contradiction that there exist some $b_i \in \mathcal{L}$, s.t. $val(b_i).ptr \neq b_{i+1}$ (or $val(b_0).prt \neq b_1$). By Lemma 9, a block b_i may appear in the sequence returned by a read operation only if it was created by a successful update operation π, on block b. Let $\mathcal{B} = \langle b_1, \ldots, b_k \rangle$ be the set of $k-1$ blocks created in π, with $b_i \in \mathcal{B}$. Let us assume w.l.o.g. that all those blocks appear in $\mathcal{L}$ as written by π (i.e., without any other blocks between any pair of them).

By the design of the algorithm, π generates a single linked path from b to b_k, by pointing b to b_1 and each b_j to b_{j+1}, for $1 \leq j < k$. Block b_k points to the block pointed by b at the invocation of π, say b'. So there exists a path $b \to b_1 \to \ldots \to b_i$ that also leads to b_i. According again to the algorithm, $b_{j+1} \in \mathcal{B}$ is created and written before b_j, for $q \leq j < k$. So when the b_j.cvr-write is invoked, the operation b_{j+1}.cvr-write has already been completed, and thus when b is written successfully all the blocks in the path are written successfully as well.

By the prefix property of the reconfiguration protocol it follows that for each b_j written by π, ρ will observe a configuration sequence $b_j.cseq_\rho$, s.t. $b_j.cseq_\pi$ is a prefix of $b_j.cseq_\rho$, and hence c_π appears in $b_j.cseq_\rho$. If c_π appears after the last finalized configuration c_ℓ in $b_j.cseq_\rho$, then the read operation will invoke c_π.get-data() and by the coverability property and property C1, will obtain a version $ver' \geq ver$. In case c_π precedes c_ℓ then a new configuration was invoked after or concurrently to π and then by Lemma 10 it follows that the version of b in c_ℓ is again $ver' \geq ver$. So we need to examine the following three cases for b_i: (i) b_i is b, (ii) b_i is b_k, and (iii) b_i is one of the blocks b_j, for $1 \leq j < k$.

Case (i): If b_i is the block b then we should examine whether $b_i.ptr \neq b_1$. Let ver the version of b written by π and ver' the version of b as retrieved by ρ. If $ver = ver'$ then ρ retrieved the block written by ω as the versions by Lemma 4 are unique. Thus, $b_i.ptr = b_1$ in this case, contradicting our assumption. In case $ver' > ver$ then there should be a successful update operation ω' that written block b with ver'. There are two cases to consider based on whether ω' introduced new blocks or not. If not then the $b.ptr = b_1$ contradicting our assumption. If it introduced a new sequence of blocks $\{b'_1, \ldots, b'_k\}$, then it should have written those blocks before writing b. In that case ρ would observe $b.ptr = b'_1$ and b'_1 would have been part of $\mathcal{L}$ which is not the case as the next block from b in $\mathcal{L}$ is b_1, leading to contradiction.

Case (ii): This case can be proven in the same way as case (i) for each block b_j, for $1 \leq j < k$.

Case (iii): If now $b_i = b_k$, then we should examine whether $b_i.ptr \neq b'$. Since b was pointing to b' at the invocation of π then b' was either (i) created during the update operation that also created b, or (ii) was created before b. In both cases b' was written before b. In case (i), by Lemma 9, the update operation that created b was successful and thus b' must be created as well. In case (ii) it follows that b is the last inserted block of an update and is assigned to point to b'. Since no block is deleted, then b' remains in $\mathcal{L}$ when b_i is created and thus b_i points to an existing block. Furthermore, since π was successful, then it successfully written b and hence only the blocks in $\mathcal{B}$ were inserted between b and b' at the response of π. In case the version of b_i was ver' and larger than the version written on b_k by π then either b_k was not extended and contains new data, or the new block is impossible as $\mathcal{L}$ should have included the blocks extending b_k. So b' must be the next block after b_i in $\mathcal{L}$ at the response of π and there is a path between b and b'. This completes the proof. ◀

We conclude with the main result of this section.

▶ **Theorem 12.** CoAresF *implements a* linearizable coverable fragmented object*.*

Proof. By the correctness proof in Section 4 follows that every block operation in CoAresF satisfies linearizable coverability and together with Lemma 11, which shows the connectivity of blocks, it follows that CoAresF implements a linearizable coverable fragmented object satisfying the properties of *fragmented linearizable coverability* (cf. Section 2). ◀

6 EC-DAP Optimization

In this section, we present an optimization in the implementation of EC-DAP, to reduce the operational latency of the read/write operations in DSMM layer. We show that this optimized EC-DAP, which we refer to as EC-DAPopt, satisfies Property 1, and thus can be used by any algorithm that utilizes the DAPs, like any variant of Ares (e.g., CoAres and CoAresF).

Description of EC-DAPopt. The main idea of the optimization is to avoid unnecessary object transmissions between the clients and the servers. Specifically, we apply the following optimization: in the get-data primitive, each server sends only the tag-value pairs with a larger or equal tag than the client's tag. In the case where the client is a reader, it performs the put-data action (propagation phase), only if the maximum tag is higher than its local one. EC-DAPopt is presented in Alg. 4 and 5. Text in blue annotates the changed or newly added code, whereas struck out blue text annotates code that has been removed from the original implementation.

Following [28], each server s_i stores a state variable, $List$, which is a set of up to $(\delta + 1)$ (tag, coded-element) pairs; δ is the maximum number of concurrent put-data operations. In EC-DAPopt, we need another two state variables, the tag of the configuration ($c.tag$) and its associated value ($c.val$). We now proceed with the details of the optimization. Note that the c.get-tag() primitive remains the same as the original.

Primitive c.get-data(). A client, during the execution of a c.get-data() primitive, queries all the servers in $c.Servers$ for their $List$, and awaits responses from $\left\lceil \frac{n+k}{2} \right\rceil$ servers. Each server generates a new list ($List'$) where it adds every (tag, coded-element) from the $List$,

Algorithm 4 EC-DAPopt implementation.

```
    at each process p_i ∈ I
2:  procedure c.get-data()
      send (QUERY-LIST, c.tag) to each s ∈ c.Servers
4:    until p_i receives List_s from each server s ∈ S_g
          ↪ s.t. |S_g| = ⌈(n+k)/2⌉
              and S_g ⊂ c.Servers
6:    ~~Tags_*^{≥k} = set of tags that appears in k lists~~
      Tags_dec^{≥k} = set of tags that appears in k lists
8:            with values
      ~~t*_max ← max Tags_*^{≥k}~~
10:   t^dec_max ← max Tags_dec^{≥k}
      ~~if t^dec_max = t*_max then~~
12:   if c.tag = t^dec_max then
        t ← c.tag
14:     v ← c.val
        return ⟨t, v⟩
16:   else if Tags_dec^{≥k} ≠ ⊥ then
        t ← t^dec_max
18:     v ← decode value for t^dec_max
        return ⟨t, v⟩
20: end procedure

    procedure c.put-data(⟨τ, v⟩))
22:   if τ > c.tag then
        code-elems = [(τ, e_1), ..., (τ, e_n)], e_i
24:            = Φ_i(v)
        send (PUT-DATA, ⟨τ, e_i⟩) to each s_i
            ↪ ∈ c.Servers
26:     until p_i receives ACK from ⌈(n+k)/2⌉ servers in
            ↪ c.Servers
        c.tag ← τ
28:     c.val ← v
    end procedure
```

Algorithm 5 The response protocols at any server $s_i \in \mathcal{S}$ in EC-DAPopt for client requests.

```
    at each server s_i ∈ S in configuration c_k
2:  State Variables:
    List ⊆ T × C_s, initially {(t_0, Φ_i(v_0))}
    Local Variables:
    List' ⊆ T × C_s, initially ⊥

4:  Upon receive (QUERY-LIST, tg_b) s_i, c_k from q
      for τ, v in List do
6:      if τ > tg_b then
          List' ← List' ∪ {⟨τ, e_i⟩}
8:      else if τ = tg_b then
          List' ← List' ∪ {⟨τ, ⊥⟩}
10:   Send List' to q
    end receive

12: Upon receive (PUT-DATA, ⟨τ, e_i⟩) s_i, c_k from q
      List ← List ∪ {⟨τ, e_i⟩}
14:   if |List| > δ + 1 then
        τ_min ← min{t : ⟨t, *⟩ ∈ List}
        /* remove the coded value */
16:     List ← List\ {⟨τ, e⟩ : τ = τ_min  ∧ ⟨τ, e⟩
                ∈ List}
18:     ~~List ← List ∪ {⟨τ_min, ⊥⟩}~~
      Send ACK to q
20: end receive
```

if the tag is higher than the $c.tag$ of the client and the (tag, $\perp$) if the tag is equal to $c.tag$; otherwise it does not add the pair, as the client already has a newer version. Once the client receives $Lists$ from $\lceil \frac{n+k}{2} \rceil$ servers, it selects the highest tag t, such that: (i) its corresponding value v is decodable from the coded elements in the lists; and (ii) t is the highest tag seen from the responses of at least k $Lists$ (see lines Alg. 4:8–10) and returns the pair (t, v). Note that in the case where any of the above conditions is not satisfied, the corresponding read operation does not complete. The main difference with the original code is that in the case where variable $c.tag$ is the same as the highest decodable tag (t^{dec}_{max}), the client already has the latest decodable version and does not need to decode it again (see line Alg. 4:12).

Primitive c.put-data($\langle t_w, v \rangle$). This primitive is executed only when the incoming t_w is greater than $c.tag$ (line Alg. 4:22). In this case, the client computes the coded elements and sends the pair $(t_w, \Phi_i(v))$ to each server $s_i \in c.Servers$. Also, the client has to update its state ($c.tag$ and $c.val$). If the condition does not hold, the client does not perform any of the above, as it already has the latest version, and so the servers are up-to-date. When a server s_i receives a message $(\text{PUT-DATA}, t_w, c_i)$, it adds the pair in its local $List$ and trims the pairs with the smallest tags exceeding the length $(\delta + 1)$ (see line Alg. 5:17).

Correctness of EC-DAPopt. We prove the following theorem.

▶ **Theorem 13** (Safety + Liveness). *EC-DAPopt satisfies both conditions of Property 1, and given that no more than δ write operations are concurrent with a read they guarantee that any operation terminates.*

The complete proof is given in Appendix B. The **main challenge** of the proof is to show that reducing the values returned by the servers does not violate linearizability, and at the same time, it does not prevent operations from reconstructing the written values, preserving liveness. We prove safety by showing that EC-DAPopt satisfies both conditions of Property 1. Particularly, we prove that the tag returned by a get-data() operation is larger than or equal to the tag written by any preceding put-data() operation, and the value returned by a get-data() operation is either written by a put-data() operation or it is the initial value of the object. Liveness is proven by showing that any put-data and get-data operation defined by EC-DAPopt terminates. In the proof, we assume an $[n, k]$ MDS code, $|c.Servers| = n$ of which no more than $\frac{n-k}{2}$ may crash, and that δ is the maximum number of put-data operations concurrent with any get-data operation. Without this assumption on δ, a get-data operation may not be able to discover a decodable value, and hence fail.

7 Experimental Evaluation

We now overview the experimental evaluation we conducted for evaluating our approach. Additional results are given in Appendix C. For a more extensive exposition of our experimental evaluation and obtained results, see [20]. The collected data are available in [3], so one could validate our analysis.

We have implemented and evaluated the following algorithms: (*i*) CoABD: the coverable version of the static ABD algorithm [27]; (*ii*) CoABDF: the fragmented version of CoABD [8]; (*iii*) CoAresABD: CoAres that uses ABD-DAP; (*iv*) CoAresABDF: fragmented CoAresABD; (*v*) CoAresEC: CoAres that uses EC-DAPopt; and (*vi*) CoAresECF: fragmented CoAresEC.

In our implementations, we consider *files*, as an example of fragmented objects. In this respect, we view a file as a linked-list of data blocks. Here, the first block, i.e., the *genesis block* b_0, is a special type of block that contains specific file information (such as the file path). For the evaluation we generate a text file with random byte strings whose size increases as the writers keep updating it. However, our implementations support any file type. The algorithms were evaluated in terms of *operational latency* and the *percentage of successful file writes*.

The experiments were executed on the emulation testbest Emulab [5], and the overlay testbed Amazon Web Services (AWS) EC2 [12]. On Emulab we used a LAN using a DropTail queue without delay or packet loss, consisting of physical nodes with one 2.4 GHz 64-bit Quad Core Xeon E5530 "Nehalem" processor and 12 GB RAM. While on AWS we used a cluster with 8 nodes of type t2.medium with 4 GB RAM, 2 vCPUs and 20 GB storage. For each experiment on Emulab we reported the average over five runs, while AWS experiments run only once.

Performance VS. Initial File Sizes. We varied the f_{size} from 1 MB to 512 MB by doubling the file size in each experimental run. The performance of some experiments is missing as the non-fragmented algorithms crashed when testing larger file sizes due to an out-of-memory error. For Emulab we used $|\mathcal{W}| = 5, |\mathcal{R}| = 5, |\mathcal{S}| = 11$, while for AWS we used $|\mathcal{W}| = 1, |\mathcal{R}| = 1, |\mathcal{S}| = 6$. Each client in Emulab performs 20 operations and in AWS 50

operations. We used a *stochastic* invocation scheme in which clients pick a random time between the interval [1...3*sec*] to invoke their next operations.

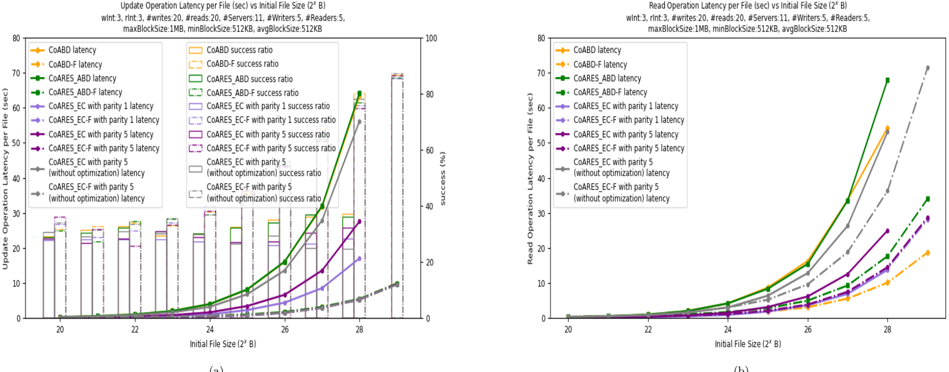


Figure 2 Emulab results for File Size experiments.

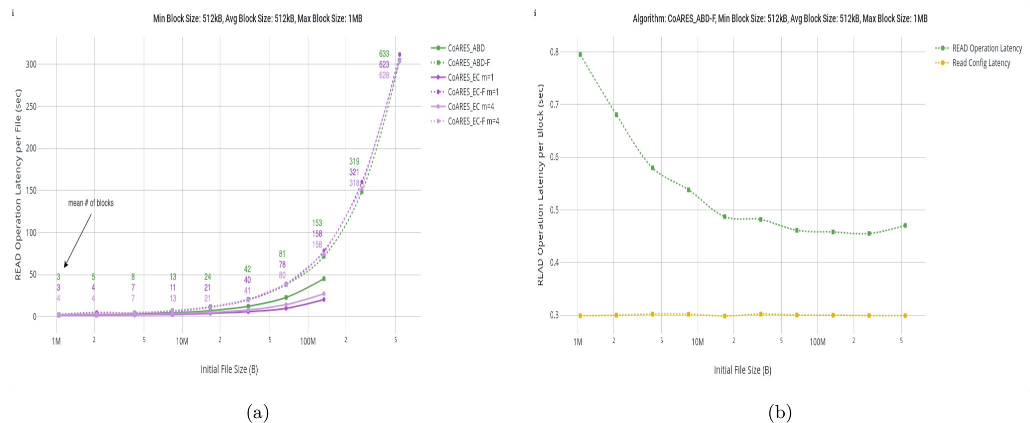


Figure 3 AWS results for File Size experiments.

Results. As shown in Fig. 2(a), the fragmented algorithms on Emulab achieve significantly smaller write latency, since the FM writes only the new and modified blocks. Also, their success ratio is higher as the file size increases, since the probability of two writes to collide on a single block decreases. The corresponding AWS findings show similar trends.

As shown in Fig. 2(b), all the fragmented algorithms on Emulab have smaller read latency than the non-fragmented ones. This happens since the readers in the shared memory level transmit only the contents of the blocks that have a newer version. On the contrary, the read latency of CoAres on AWS (Fig. 3(a)) has not improved with the fragmentation strategy. The CoAresF operations perform at least two additional rounds (compared to CoABDF), in order to read the configuration before each of the two phases. Thus, when the FM module sends multiple read block requests, has a significant stable overhead for each block request in the real network conditions of AWS (Fig. 3(b)).

We can also observe from the Figs. 2(a)-(b), 3(a) that the further increase of the parity (m) of CoAresEC and CoAresECF algorithms (and thus higher fault-tolerance) the larger the latency. In addition, the read and write latency of these algorithms when used with EC-DAP are double than of the ones when our optimized DAP (EC-DAPopt) is used.

Trade-offs. During the deployment, the main trade-offs we have identified are the following:

Block size of FM. The performance of data striping highly depends on the block size. There is a trade-off between splitting the object into smaller blocks, for improving the concurrency in the system, and paying for the cost of sending these blocks in a distributed fashion. Therefore, it is crucial to discover the "golden" spot with the minimum communication delays (while having a large block size) that will ensure a small expected probability of collision (as a parameter of the block size and the delays in the network).

Parity of EC. There is a trade-off between operation latency and fault-tolerance in the system: the further increase of the parity (and thus higher fault-tolerance) the larger the latency.

Parameter δ of EC. The value of δ is equal to the number of writers. As a result, as the number of writers increases, the latency of the first phase of EC also increases, since each server sends the list with all the concurrent values. In this point, we can understand the importance of the optimization (EC-DAPopt) in the DSMM layer.

8 Conclusions

In this paper we have presented and rigorously proved correct CoAresF, the first dynamic distributed shared memory that utilizes coverable fragmented objects and enables the use of erasure coding. To achieve this, we developed a coverable version of Ares and integrated it with CoBFS. When CoAresF is used with an (optimized) Erasure Coded DAP we obtain a two-level striping dynamic and robust distributed shared memory system providing strong consistency and high access concurrency to large objects (e.g., files). We have complemented our development with an extensive experimental evaluation over the Emulab and AWS testbeds. Compared to the approach that does not use the fragmentation layer of CoBFS (CoAres), CoAresF is optimized with an efficient access to shared data under heavy concurrency. For future work, we plan to explore how to reduce the overhead of read operations. In addition, as our service achieves highly scalable performance, it seems suitable for a P2P environment; any physical node could serve both as a client and a data host.

References

1 Cassandra. `https://cassandra.apache.org/_/index.html`.
2 Colossus. `https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system`.
3 Data repository. `https://github.com/atrigeorgi/fragmentedARES-data.git`.
4 Dropbox. `https://www.dropbox.com/`.
5 Emulab network testbed. `https://www.emulab.net/`.
6 Hdfs. `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`.
7 M.K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. In *Proceedings of the 28th ACM symposium on Principles of distributed computing (PODC '09)*, pages 17–25, New York, NY, USA, 2009. ACM.
8 A.F. Anta, C. Georgiou, T. Hadjistasi, E. Stavrakis, and A. Trigeorgi. Fragmented Object : Boosting Concurrency of Shared Large Objects. *In Proc.of SIROCCO*, pages 1–18, 2021.
9 Antonio Fernández Anta, Chryssis Georgiou, Theophanis Hadjistasi, Nicolas Nicolaou, Efstathios Stavrakis, and Andria Trigeorgi. Fragmented objects: Boosting concurrency of sharedlarge objects. *CoRR*, abs/2102.12786, 2021. `arXiv:2102.12786`.
10 H. Attiya. Robust Simulation of Shared Memory: 20 Years After. *Bulletin of the EATCS*, 100:99–114, 2010.
11 H. Attiya, A. Bar-Noy, and D. Dolev. Sharing Memory Robustly in Message-Passing Systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

12 AWS EC2. https://aws.amazon.com/ec2/.

13 Paul Black. Ratcliff pattern recognition. *Dictionary of Algorithms and Data Structures*, 2021.

14 A. Carpen-amarie. BlobSeer as a Data-Storage Facility for Clouds: Self-Adaptation, Integration, Evaluation, PhD Thesis, France, 2012.

15 P. Dutta, R. Guerraoui, R.R. Levy, and A. Chakraborty. How fast can a distributed atomic read be? *In Prof. of PODC*, pages 236–245, 2004.

16 Rui Fan and Nancy A. Lynch. Efficient replication of large data objects. In Faith Ellen Fich, editor, *Distributed Computing, 17th International Conference, DISC 2003, Sorrento, Italy, October 1-3, 2003, Proceedings*, volume 2848 of *Lecture Notes in Computer Science*, pages 75–91. Springer, 2003. doi:10.1007/978-3-540-39989-6_6.

17 E. Gafni and D. Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9363:140–153, 2015. doi:10.1007/978-3-662-48653-5_10.

18 C. Georgiou, N. Nicolaou, and A.A. Shvartsman. Fault-tolerant semifast implementations of atomic read/write registers. *Journal of Parallel and Distributed Computing*, 69(1):62–79, 2009.

19 Chryssis Georgiou, Theophanis Hadjistasi, Nicolas Nicolaou, and Alexander A. Schwarzmann. Implementing three exchange read operations for distributed atomic storage. *J. Parallel Distributed Comput.*, 163:97–113, 2022. doi:10.1016/j.jpdc.2022.01.024.

20 Chryssis Georgiou, Nicolas Nicolaou, and Andria Trigeorgi. Fragmented ARES: dynamic storage for large objects. *CoRR*, abs/2201.13292, 2022. arXiv:2201.13292.

21 Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *The Google File System*, 53(1):79–81, 2003.

22 Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. RAMBO: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Comput.*, 23(4):225–272, 2010. doi:10.1007/s00446-010-0117-1.

23 Vincent Gramoli, Nicolas Nicolaou, and Alexander A. Schwarzmann. *Consistent Distributed Storage*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2021. doi:10.2200/S01069ED1V01Y202012DCT017.

24 M.P. Herlihy and J.M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

25 L. Jehl, R. Vitenberg, and H. Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *International Symposium on Distributed Computing*, pages 154–169. Springer, 2015.

26 N.A. Lynch and A.A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. *In Proc. of FTCS*, pages 272–281, 1997.

27 N. Nicolaou, A.F. Anta, and C. Georgiou. Cover-ability: Consistent versioning in asynchronous, fail-prone, message-passing environments. In *Proc. of IEEE NCA 2016*, pages 224–231. Institute of Electrical and Electronics Engineers Inc., 2016. doi:10.1109/NCA.2016.7778622.

28 Nicolas Nicolaou, Viveck Cadambe, N. Prakash, Andria Trigeorgi, Kishori M. Konwar, Muriel Medard, and Nancy Lynch. ARES: Adaptive, Reconfigurable, Erasure coded, Atomic Storage. *ACM Transactions on Storage (TOS)*, 2022. Accepted. Also in arXiv:1805.03727.

29 Satadru Pan, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, J R Tipton, Theano Stavrinos, Ethan Katz-Bassett, and Wyatt Lloyd. Facebook's Tectonic Filesystem: Efficiency from Exascale, 2021. URL: https://www.usenix.org/conference/fast21/presentation/pan.

30 M O Rabin. Fingerprinting by random polynomials, 1981. URL: http://www.xmailserver.org/rabin.pdf.

31 Irving S. Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of The Society for Industrial and Applied Mathematics*, 8:300–304, 1960.

32 M.V. Steen and A.S. Tanenbaum. *Distributed Systems, 3rd ed.* distributed-systems.net, 2017.

33 A. Tridgell and P. Mackerras. The rsync algorithm. *Imagine*, 1996.

34 P. Viotti and M. Vukolic. Consistency in non-transactional distributed storage systems. *ACM Computing Surveys (CSUR)*, 49:1–34, 2016.

A Our and Prior Work: A Comparative Table

Table 1 presents a comparison of the main characteristics of the distributed algorithms and storage systems. As we can see, systems that use relaxed or eventual consistency have serious issues when conflicting writes appear. Others guarantee strong consistency but have centralized components. To our opinion the most appropriate model able to provide high consistency, concurrency and availability seems to be the Atomic / Linearizability Consistency model. Some previous attempts, such as LDR [16], were promising, but they seem to suffer from communication delays and communication overheads since the whole object is still transmitted in every message exchanged between the clients and the replica servers. Also, the table shows well-known algorithms for reconfigurable atomic storage.

B Correctness of EC-DAPopt

To prove the correctness of EC-DAPopt, we need to show that it is *safe*, i.e., it ensures the necessary Property 1, and *live*, i.e., it allows each operation to terminate. In the following proof, we will not refer to the get-tag access primitive that the EC-DAP algorithm uses [28], as the optimization has no effect on this operation, so it should preserve safety as shown in [28].

For the following proofs we fix the configuration to c as it suffices that the DAPs preserve Property 1 in any single configuration. Also we assume an $[n, k]$ MDS code, $|c.Servers| = n$ of which no more than $\frac{n-k}{2}$ may crash, and that δ is the maximum number of put-data operations concurrent with any get-data operation.

We first prove Property 1-C2 as it is later being used to prove Property 1-C1.

▶ **Lemma 14** (C2). *Let ξ be an execution of an algorithm A that uses the* EC-*DAPopt. If ϕ is a* c.get-data() *that returns $\langle \tau_\pi, v_\pi \rangle \in \mathcal{T} \times \mathcal{V}$, then there exists π such that π is a* c.put-data($\langle \tau_\pi, v_\pi \rangle$) *and ϕ did not complete before the invocation of π. If no such π exists in ξ, then (τ_π, v_π) is equal to (t_0, v_0).*

Proof. It is clear that the proof of property $C2$ of EC-DAPopt is identical with that of EC-DAP. This happens as the initial value of the *List* variable in each servers s in $\mathcal{S}$ is still $\{(t_0, \Phi_s(v_\pi))\}$, and the new tags are still added to the *List* only via put-data operations. Thus, each server during a get-data operation includes only written tag-value pairs from the *List* to the *List'*. ◀

▶ **Lemma 15** (C1). *Let ξ be an execution of an algorithm A that uses the* EC-*DAPopt. If ϕ is* c.put-data($\langle \tau_\phi, v_\phi \rangle$)*, for $c \in \mathcal{C}$, $\langle \tau_\phi, v_\phi \rangle \in \mathcal{T} \times \mathcal{V}$, and π is* c.get-data() *that returns $\langle \tau_\pi, v_\pi \rangle \in \mathcal{T} \times \mathcal{V}$ and $\phi \to \pi$ in ξ, then $\tau_\pi \geq \tau_\phi$.*

Proof. Let p_ϕ and p_π denote the processes that invoke ϕ and π in ξ. Let $S_\phi \subset \mathcal{S}$ denote the set of $\lceil \frac{n+k}{2} \rceil$ servers that responds to p_ϕ, during ϕ, and by S_π the set of $\lceil \frac{n+k}{2} \rceil$ servers that responds to p_π, during π.

Per Alg. 5:13, every server $s \in S_\phi$, inserts the tag-value pair received in its local *List*. Note that once a tag is added to *List*, its associated tag-value pair will be removed only when the *List* exceeds the length $(\delta + 1)$ and the tag is the smallest in the *List* (Alg. 5:14–17).

Table 1 Comparative table of distributed algorithms and storage systems.

Algorithm/ System	Data scalability	Data access Concurrency	Consistency guarantees	Versioning	Data Striping	Non-blocking Reconfiguration
GFS [21]	YES	concurrent appends	relaxed	YES	YES	YES (short downtime)
HDFS [6]	YES	files restrict one writer at a time	strong (centralized)	NO	YES	YES
Cassandra [1]	YES	YES	tunable (default= eventual)	YES	NO	NO
Dropbox [4]	YES	creates conflicting copies	eventual	YES	YES	N/A
Colossus [2]	YES	concurrent appends	relaxed	YES	YES	YES
Blobseer [14]	YES	YES	strong (centralized)	YES	YES	YES
Tectonic [29]	YES	files restrict one writer at a time	strong	YES	YES	YES
CoABD [27]	NO	YES	strong	YES	NO	NO
CoBFS [8]	YES	YES	strong	YES	YES	NO
RAMBO [22]	NO	NO	strong	NO	NO	YES
DynaStore [7]	NO	NO	strong	NO	NO	YES
SM-Store [25]	NO	NO	strong	NO	NO	YES
SpSnStore [17]	NO	NO	strong	NO	NO	YES
AresABD [28]	NO	NO	strong	NO	NO	YES
AresEC [28]	NO	NO	strong	NO	YES	YES
CoAresABD [our work]	NO	NO	strong	YES	NO	YES
CoAresEC [our work]	NO	NO	strong	YES	YES	YES
CoAresABDF [our work]	YES	YES	strong	YES	YES	YES
CoAresECF [our work]	YES	YES	strong	YES	YES (2 striping methods)	YES

When replying to π, each server in S_π includes a tag in $List'$, only if the tag is larger or equal to the tag associated to the last value decoded by p_π (lines Alg. 5:6–9). Notice that as $|S_\phi| = |S_\pi| = \lceil \frac{n+k}{2} \rceil$, the servers in $|S_\phi \cap S_\pi| \geq k$ reply to both π and ϕ. So there are two cases to examine: (a) the pair $\langle \tau_\phi, v_\phi \rangle \in Lists'$ of at least k servers $S_\phi \cap S_\pi$ replied to π, and (b) the $\langle \tau_\phi, v_\phi \rangle$ appeared in fewer than k servers in S_π.

Case a: In the first case, since π discovered τ_ϕ in at least k servers it follows by the algorithm that the value associated with τ_ϕ will be decodable. Hence $t^{dec}_{max} \geq \tau_\phi$ and $\tau_\pi \geq \tau_\phi$.

Case b: In this case τ_ϕ was discovered in less than k servers in S_π. Let τ_ℓ denote the last tag returned by p_π. We can break this case in two subcases: (i) $\tau_\ell > \tau_\phi$, and (ii) $\tau_\ell \leq \tau_\phi$.

In case (i), no $s \in S_\pi$ included τ_ϕ in $List'_s$ before replying to π. By Lemma 14, the c.put-data$(\langle \tau_\ell, * \rangle)$ was invoked before the completion of the $*$.get-data() operation from p_π that returned τ_ℓ. It is also true that p_π discovered $\langle \tau_\ell, * \rangle$ in more than k servers since it managed to decode the value. Therefore, in this case $t^{dec}_{max} \geq \tau_\ell$ and thus $\tau_\pi > \tau_\phi$.

In case (ii), a server $s \in S_\phi \cap S_\pi$ will not include τ_ϕ iff $|Lists'_s| = \delta + 1$, and therefore the local $List$ of s removed τ_ϕ as the smallest tag in the list. According to our assumption though, no more than δ put-data operations may be concurrent with a get-data operation. Thus, at least one of the put-data operations that wrote a tag $\tau' \in Lists'_s$ must have completed before π. Since τ' is also written in $|S'| = \frac{n+k}{2}$ servers then $|S_\pi \cap S'| \geq k$ and hence π will be able to decode the value associated to τ', and hence $t^{dec}_{max} \geq \tau_\ell$ and $\tau_\pi > \tau_\phi$, completing the proof of this lemma. ◀

▶ **Theorem 16** (Safety). *Let ξ be an execution of an algorithm A that contains a set Π of complete* get-data *and* put-data *operations of Algorithm 4. Then every pair of operations $\phi, \pi \in \Pi$ satisfy Property 1.*

Proof. Follows directly from Lemmas 14 and 15. ◀

Liveness requires that any put-data and get-data operation defined by EC-DAPopt terminates. The following theorem captures the main result of this section.

▶ **Theorem 17** (Liveness). *Let ξ be an execution of an algorithm A that utilises the* EC-*DAPopt. Then any* put-data *or* get-data *operation π invoked in ξ will eventually terminate.*

Proof. Given that no more than $\frac{n-k}{2}$ servers may fail, then from Algorithm 4 (lines Alg. 4:21–29), it is easy to see that there are at least $\frac{n+k}{2}$ servers that remain correct and reply to the put-data operation. Thus, any put-data operation completes.

Now we prove the liveness property of any get-data operation π. Let p_ω and p_π be the processes that invoke the put-data operation ω and get-data operation π. Let S_ω be the set of $\lceil \frac{n+k}{2} \rceil$ servers that responds to p_ω, in the put-data operations, in ω. Let S_π be the set of $\lceil \frac{n+k}{2} \rceil$ servers that responds to p_π during the get-data step of π. Note that in ξ at the point execution T_1, just before the execution of π, none of the write operations in Λ is complete. Let T_2 denote the earliest point of time when p_π receives all the $\lceil \frac{n+k}{2} \rceil$ responses. Also, the set Λ includes all the put-data operations that starts before T_2 such that $tag(\lambda) > tag(\omega)\}$. Observe that, by algorithm design, the coded-elements corresponding to t_ω are garbage-collected from the $List$ variable of a server only if more than δ higher tags are introduced by subsequent writes into the server. Since the number of concurrent writes $|\Lambda|$, s.t. $\delta > |\Lambda|$ the corresponding value of tag t_ω is not garbage collected in ξ, at least until execution point T_2 in any of the servers in S_ω. Therefore, during the execution fragment between the execution points T_1 and T_2 of the execution ξ, the tag and coded-element pair is present in the $List$ variable of every server in S_ω that is active. As a result, the tag

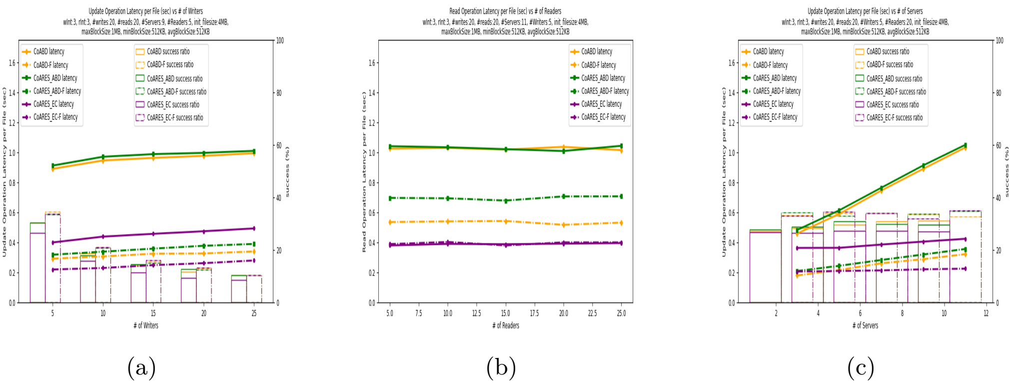


Figure 4 Emulab results for Scalability experiments.

and coded-element pairs, $(t_\omega, \Phi_s(v_\omega))$ exists in the $List$ received from any $s \in S_\omega \cap S_\pi$ during operation π. Note that since $|S_\omega| = |S_\pi| = \lceil \frac{n+k}{2} \rceil$ hence $|S_\omega \cap S_\pi| \geq k$ and hence $t_\omega \in Tags^{\geq k}_{dec}$, the set of decode-able tag, i.e., the value v_ω can be decoded by p_π in π, which demonstrates that $Tags^{\geq k}_{dec} \neq \emptyset$.

Next we want to argue that t^{dec}_{max} is the maximum tag that π discovers via a contradiction: we assume a tag t_{max}, which is the maximum tag π discovers, but it is not decode-able, i.e., $t_{max} \notin Tags^{\geq k}_{dec}$ and $t_{max} > t^{dec}_{max}$. Let $S^k_\pi \subset S$ be any subset of k servers that responds with t_{max} in their $List'$ variables to p_π. Note that since $k > n/3$ hence $|S_\omega \cap S^k_\pi| \geq \lceil \frac{n+k}{2} \rceil + \lceil \frac{n+1}{3} \rceil \geq 1$, i.e., $S_\omega \cap S^k_\pi \neq \emptyset$. Then t_{max} must be in some servers in S_ω at T_2 and since $t_{max} > t^{dec}_{max} \geq t_\omega$. Now since $|\Lambda| < \delta$ hence $(t_{max}, \Phi_s(v_{max}))$ cannot be removed from any server at T_2 because there are not enough concurrent write operations (i.e., writes in Λ) to garbage-collect the coded-elements corresponding to tag t_{max}. Also since π cannot have a local tag larger than t_{max}, according to the lines Alg. 5:6–9 each server in S_π includes the t_{max} in its replies. In that case, t_{max} must be in $Tag^{\geq k}_{dec}$, a contradiction. ◀

C Additional Experimental Results

C.1 Performance VS. Scalability of Nodes Under Concurrency

This scenario is constructed to compare the read, write and recon latency of the algorithms, as the number of service participants increases.

Without Reconfiguration. In both Emulab and AWS, we varied the number of readers $|\mathcal{R}|$ and the number of writers $|\mathcal{W}|$ from 5 to 25, while the number of servers $|\mathcal{S}|$ varies from 3 to 11. In AWS, the clients and servers are distributed in a round-robin fashion. We calculate all possible combinations of readers, writers and servers where the number of readers or writers is kept to 5. In total, each writer performs 20 writes and each reader 20 reads. The size of the file used is 4 MB. The maximum, minimum and average block sizes were set to 1 MB, 512 kB and 512 kB respectively. To match the fault-tolerance of ABD-BASED algorithms, we used a different parity for EC-BASED algorithms (except in the case of 3 servers to avoid replication). With this, the EC client has to wait for responses from a larger quorums. The parity value of the EC-BASED algorithms is set to $m = 1$ for $|\mathcal{S}| = 3$, $m = 2$ for $|\mathcal{S}| = 5$, $m = 3$ for $|\mathcal{S}| = 7$, $m = 4$ for $|\mathcal{S}| = 9$ and $m = 5$ for $|\mathcal{S}| = 11$.

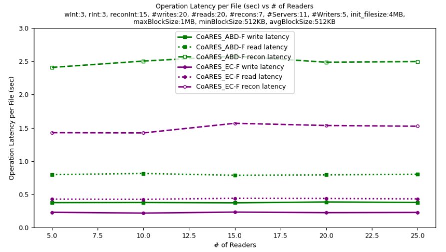


Figure 5 Emulab results when Reconfiguring to the Same DAP_s.

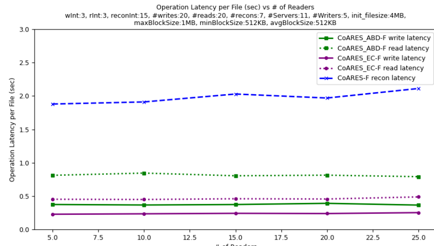
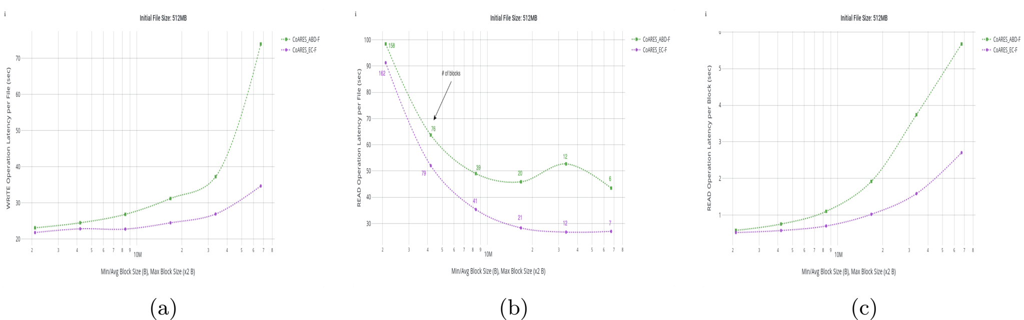


Figure 6 Emulab results when Reconfiguring DAP_s Randomly.

Results. The results obtained in this scenario are presented in Fig. 4. As expected, CoAresEC has the lowest update latency among non-fragmented algorithms because of the striping level. Each object is divided into k encoded fragments that reduce the communication latency (since it transfers less data over the network) and the storage utilization. The fragmented algorithms perform significantly better update latency than the non-fragmented ones, even when the number of writers increases (see Fig. 4(a)).This is because the non-fragmented writer updates the whole file, while each fragmented writer updates only a subset of blocks. We observe that the update operation latency in algorithms CoABD and CoAresABD increases as the number of servers increases, while the operation latency of CoAresEC decreases or stays the same (Figs. 4(c)) That is because when increasing the number of servers, the quorum size grows but the message size decreases. Therefore, while both non-fragmented ABD-based algorithms and CoAresEC waits for responses the decreased message size. When going from 7 to 9 servers, we observe a decrease in latency. This is due the choice of parity value (parameter of EC-based algorithms) that we select for 7 servers. Due to the block allocation strategy in fragment algorithms, more data are successfully written (cf. Fig. 4(a), 4(b)), explaining the slower CoAresF read operation (cf. Figs. 4(b)). The corresponding AWS findings show similar trends.

With Reconfiguration. We built four extra experiments in Emulab to verify the correctness of the variants of Ares when reconfigurations coexist with read/write operations. The experiments differ in the way the reconfigurer works; three experiments use $|\mathcal{S}| = 11$ and are based on the way the reconfigurer chooses the next storage algorithm (i.e., two reconfiguring to the same DAP and one reconfiguring to a random DAP); one in which the reconfigurer changes the storage algorithm and the quorum of servers. In the latter scenario the reconfigurer chooses randomly between $[3, 5, 7, 9, 11]$ servers. All experiments run on CoAres and CoAresF use one reconfigurer.

Results. Due to space limit, we report only one of the experiments (all results can be found in [20]). As we mentioned earlier, our choice of k minimizes the coded fragment size but introduces bigger quorums and thus larger communication overhead. As a result, in smaller file sizes, Ares (either fragmented or not) may not benefit so much from the coding, bringing the delays of the CoAresEC and CoAresABD closer to each other (cf. Fig. 5). However, the read latency of CoAresECF is significant lower than of CoAresABDF. This is because the CoAresECF takes less time to transfer the blocks to the new configuration.



(a) (b) (c)

Figure 7 AWS results for Min/Avg/Max Block Sizes' experiments.

C.2 Performance VS. Block Sizes

This scenario evaluates how the block size impacts the latencies when having a rather large file size. We varied the minimum and average b_{sizes} from 2 MB to 64 MB and the maximum b_{size} from 4 MB to 1 GB. In total, each writer performs 20 writes and each reader 20 reads. The size of the initial file used was set to 512 MB.

Emulab parameters: $|\mathcal{W}| = 5, |\mathcal{R}| = 5, |\mathcal{S}| = 11$. For EC-BASED algorithms, $m = 1$ and the quorum size is 11. For ABD-BASED algorithms we used quorums of size 4.

AWS parameters: $|\mathcal{W}| = 1, |\mathcal{R}| = 1, |\mathcal{S}| = 6$. For EC-BASED algorithms, $m = 1$ and the quorum size is 6. For ABD-BASED algorithms we used quorums of size 4.

Results. As all examined block sizes are enough to fit the text additions no new blocks are created. All the algorithms achieve the maximal update latency as the block size gets larger (Fig 7(a)). CoAresECF has the lower impact as block size increases mainly due to the extra level of striping. Similar behaviour has the read latency in Emulab. However, in real time conditions of AWS, the read latency of a higher number of relatively large blocks (Fig. 7(c)) has a significant impact on overall latency, resulting in a larger read latency (Fig. 7(b)).

Fast Distributed Vertex Splitting with Applications

Magnús M. Halldórsson
Reykjavik University, Iceland

Yannic Maus
TU Graz, Austria

Alexandre Nolin
Reykjavik University, Iceland[1]
CISPA, Saarbrücken, Germany

Abstract

We present poly log log n-round randomized distributed algorithms to compute vertex splittings, a partition of the vertices of a graph into k parts such that a node of degree $d(u)$ has $\approx d(u)/k$ neighbors in each part. Our techniques can be seen as the first progress towards general poly log log n-round algorithms for the Lovász Local Lemma.

As the main application of our result, we obtain a randomized poly log log n-round CONGEST algorithm for $(1+\varepsilon)\Delta$-edge coloring n-node graphs of sufficiently large constant maximum degree Δ, for any $\varepsilon > 0$. Further, our results improve the computation of defective colorings and certain tight list coloring problems. All the results improve the state-of-the-art round complexity exponentially, even in the LOCAL model.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Graph problems, Edge coloring, List coloring, Lovász local lemma, LOCAL model, CONGEST model, Distributed computing

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.26

Related Version *Full Version*: https://arxiv.org/abs/2208.08119

1 Introduction

Consider the following fundamental load-balancing problem: Partition the vertices of an n-node degree-Δ graph into two parts so that each node has at most $(1+\varepsilon)\Delta/2$ neighbors in each part, where $\varepsilon > 0$ is an arbitrary given constant. When Δ is large enough (say, superlogarithmic), such a *2-splitting* is trivially achieved w.h.p. without communication. Can it be solved fast distributively for arbitrary Δ?

The 2-splitting problem can be formulated as an instance of the *Lovász local lemma* (LLL). Consider some *"bad"* events over a probability space. The celebrated Lovász local lemma states that if the events satisfy certain limited dependencies, then there is a positive probability that none of them happens [16]. In the 2-splitting problem, the probability space is spanned by each node picking a part uniformly at random and there is a bad event for each node that occurs when too many of its neighbors are in any one of the parts. In the constructive version of the LLL, the objective is to also compute an assignment avoiding all bad events, and using known distributed LLL algorithms, it can be solved in $O(\log n)$ distributed rounds [41, 13]. For small Δ, there is a faster $O(\Delta^2 + \text{poly}\log\log n)$-round algorithm [18], but it does not improve the case of arbitrary Δ. This leaves a major open problem: Can we close the gap between the $O(\log n)$ upper bound and $\Omega(\log_\Delta \log n)$ lower bound [2]? Clarifying this for 2-splitting would be the first step towards resolving the complexity of general distributed LLLs.

[1] Alexandre Nolin changed affiliation from Reykjavik University to CISPA after this work was completed.

© Magnús M. Halldórsson, Yannic Maus, and Alexandre Nolin;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 26; pp. 26:1–26:24
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Our central technical result is to answer this question affirmatively, even in the bandwidth-constrained CONGEST model, by giving a $\text{poly}\log\log n$-round algorithms for 2-splitting and various other vertex- and edge splitting problems. Note that these are also exponential improvements for the LOCAL model. Such splitting problems are pervasive in distributed graph algorithmics [27, 28, 17, 19, 25, 23, 31, 2]. They can be viewed as questions of rounding and discrepancy, and they are frequently the major building block in solving various classic problems when using a divide-and-conquer approach.

We illustrate the reach of the techniques by giving much faster algorithms for two classic coloring problems.

▶ **Theorem 1** (Edge coloring). *For any constant $\varepsilon > 0$, there is a* $\text{poly}\log\log n$*-round randomized* CONGEST *algorithm to compute a $(1+\varepsilon)\Delta$-edge coloring on any graph with maximum degree $\Delta \geq \Delta_0$ where Δ_0 is a sufficiently large constant.*

Notice that as a function of n alone, previous methods use at least $\Omega(\log n)$ time, even in the LOCAL model. The problem has a $\Omega(\log_\Delta \log n)$ lower bound [10]. Previously, $\text{poly}\log\log n$-round algorithms were only known for $2\Delta - 1$-edge coloring [15, 29], and $O(\log_\Delta n + \text{poly}\log\log n)$-round algorithms for using any smaller number of colors [14, 42, 13, 15, 10], even in the LOCAL model. Tackling this problem in CONGEST is non-trivial as it depends on LLL, which only has efficient known CONGEST solutions for the constant-degree case [39].

In the second application, the (L,T)*-list coloring* problem, each node of a graph is given a list of at least L colors such that any color in its list appears in at most T neighbors' lists. We ask for a valid node coloring where each node receives a color from its list, with the ratio L/T as small as possible. Observe that the degree of a node can be much larger than its list of colors, and thus greedy approaches are insufficient, even centrally.

▶ **Theorem 2** (List coloring). *There is a* $\text{poly}\log\log n$*-round randomized* LOCAL *algorithm for the list coloring problem, for any T and L with $L \geq (1+\delta)T$, for any $\delta > 0$ and any $\Delta \geq \Delta_0$, for some absolute constant Δ_0.*

Previous algorithms either used $O(\log n)$-rounds [13] or required $L/T \geq C_0$ for a (large) constant C_0 [18]. See Appendix B for more related work on list coloring.

1.1 Contributions on Splitting problems

The main ingredient for both of the above applications is our efficient method to split graphs into small degree subgraphs. A *k-vertex splitting problem* with *discrepancy z* is a partition of the vertex set into k parts $V_1, \ldots, V_k$ such that, for each $i \in [k]$, each node $v \in V$ has $d(v)/k \pm z$ neighbors in V_i. Intuitively, splitting a graph into k parts with a discrepancy of $\varepsilon\Delta/k$ is useful to solve various problems that are easier on low-degree graphs. These problems must be resilient to imperfect splits, which is ensured in coloring problems by having a surplus of colors.

A Chernoff bound argument shows that such splittings are quite easy for high degrees ($\Delta \gg k \log n$). We obtain the following theorem:

▶ **Theorem 3.** *There exists a universal constant $c_1 > 0$ s.t.: For any $\varepsilon > 0$, maximum degree $\Delta \leq \text{poly}\log n$, and $k \leq c_1 \cdot (\varepsilon^4\Delta/\ln\Delta)$, there is a distributed randomized* LOCAL *algorithm to compute a k-vertex splitting with discrepancy $\varepsilon\Delta/k$ in $O(1/\varepsilon) + \text{poly}\log\log n$ rounds.*

The $\text{poly}\log\log n$ term in the runtime of Theorem 3 stems from solving LLL instances of size $N = \text{poly}\log n$ deterministically. Any improvement on such algorithms immediately carries over to our result. However, there is a lower bound of $\Omega(\log_\Delta \log n)$ rounds for randomized and $\Omega(\log_\Delta n)$ rounds for deterministic algorithms for the respective splitting problems (and hence also for the LLL problem) [2]. These lower bounds even hold for a weak variant of the vertex splitting problem, in which each node only needs to have one neighbor of each color class.

Variants. Our main applications require subtle variations of the splitting problem. To this end, we solve a more general problem, where we separate the two functions of each node: as a variable (which part is it assigned to) and as an event (whether its neighborhood is evenly split). In the *bipartite k-vertex splitting problem with discrepancy z*, we have a set V^{L} of nodes for the events and a set V^{R} of nodes for the variables, with an edge between every dependent variable-edge pair. We wish to partition V^{R} into k parts such that each event vertex $u \in V^{\mathsf{L}}$ has $d(u)/k \pm z$ neighbors in each part.

▶ **Theorem 4.** *There exists a universal constant $c_2 > 0$ s.t.: For any $\varepsilon > 0$, maximum degree $\Delta \leq \text{poly}\log n$ and $k \leq c_2 \cdot (\varepsilon^4 \Delta_{\mathsf{L}} / \ln \Delta)$, there is a distributed randomized* LOCAL *algorithm to compute a bipartite k-vertex splitting problem with discrepancy $\varepsilon\Delta_{\mathsf{L}}/k$ in $O(1/\varepsilon)+ \text{poly}\log\log n$ rounds.*

We also devise CONGEST versions of Theorems 3 and 4 that are essential to our edge coloring result in CONGEST. The formal statement appears in Theorem 20 and requires k to be a $O(\log^2 \log n)$ factor smaller than in Theorems 3 and 4.

In a *d-defective c-coloring*, each of the c color classes induces a graph of maximum degree d. Defective colorings are frequently used in divide-and-conquer approaches to other coloring problems [34, 3] and they have been studied in several works, e.g., [34, 3, 33, 26, 37], usually stating variations of deterministic algorithms for computing d-defective coloring with $O((\Delta/d)^2)$ colors. As any vertex splitting is also a defective coloring, Theorem 3 implies a $\text{poly}\log\log n$-round algorithm for $(1+\varepsilon)\Delta/k$-defective k-coloring. Previous algorithms for $(1+\varepsilon)\Delta/k$ defective colorings either used $O(k^2)$ colors [34, 37] or a logarithmic number of rounds through solving the respective LLL problem [13].

1.2 Challenges to Fast and Efficient Splitting

Known approaches to splitting (or any of the other problems we consider) all build on the *Lovász Local Lemma* (LLL) for the low-degree case ($\Delta < \log n$). This hits a wall, since there are no strongly sublogarithmic time distributed LLL algorithms known, in spite of intensive efforts [12, 10].

There are two known approaches to distributed LLL algorithms. The breakthrough Moser-Tardos method [41, 13] is based on stochastic local search, which appears to inherently require logarithmic rounds. The other approach is to use the *shattering* technique, solving most of the problem quickly, leading to small remaining "shattered" subgraphs for which we can afford to apply slower techniques. This was introduced by Beck [5] in the centralized setting and Alon [1] in the parallel setting.

Fischer and Ghaffari [18] proposed a shattering-based distributed algorithm, modeled on an earlier sequential algorithm of Molloy and Reed [40]. Using recent network decompositions [46], their method runs in $O(\Delta^2 + \text{poly}\log\log n)$ time, which is fast for low-degree graphs ($\Delta \leq \text{poly}\log\log n$) but doesn't improve the general case. To understand the issue, let us examine more closely the reasoning behind the method of [18] in the context of 2-splitting.

A random assignment (of the nodes into the parts) is easily seen to satisfy the lion's share of the vertices, where "satisfied" means having discrepancy within the stated bound. Each node is so likely to be satisfied that the remaining subgraph is indeed *shattered*: the connected components induced by the set of unsatisfied nodes are of small size (assuming $\Delta \leq \text{poly}\log n$, which is the hard case). One natural approach is to *undo* the assignment to the unsatisfied nodes, and then solve the problem separately on the unsatisfied nodes. However, this causes new problems: Nodes that were previously satisfied may become hard to satisfy. For instance, suppose a node v has neighbors $u_1, \ldots, u_t$ assigned to the first part and nodes $u_{t+1}, \ldots, u_{2t}$ assigned to the second part, for a perfect split. But it is now possible that all of $u_1, u_2, \ldots, u_t$ are retracted, having their assignment undone. Then, satisfying v now requires assigning its neighbors back to the second part, which leaves little flexibility, and there may not be a valid solution.

Fischer and Ghaffari [18] (and [40]) fix this by sampling the random variables (i.e., which part each node is assigned to) only *gradually*, i.e., at most one variable per event is sampled simultaneously. Along with "freezing" (or deferring) certain nodes, this ensures that no vertex experiences too heavy a setback caused by retractions. The gradual sampling is achieved by first computing a distance-2 coloring of the graph using $O(\Delta^2)$ colors, and then sampling only the nodes of a single color class at a time. The downside is that this unavoidably requires time complexity at least Δ^2.

A different type of challenge appears when aiming for bandwidth-efficient algorithms. Even if one drastically improves upon the sketched $O(\Delta^2)$ "pre-shattering" procedure from [18], the deterministic procedure used in the "post-shattering" phase of their algorithm to complete the obtained partial solution makes heavy use of the unlimited bandwidth of the LOCAL model. In fact, while both types of randomized distributed LLL methodologies [38, 13] are themselves frugal in terms of bandwidth, known deterministic LLL algorithms are based on bandwidth-hungry generic derandomization results [27, 23, 46].

1.3 Our Methods in a Nutshell

Fast splitting. Our approach is to sample gradually – like in [40, 18] – but faster. We group the variables (representing the part assigned to a node) into *buckets* and then sample the variables one bucket at a time. This is crucially done so that the impact of any given bucket on any given event is limited (namely, the number of neighbors of a node in any given bucket is upper bounded), so that we can recover from bad probabilistic assignments. Intuitively, a node might have to "give up" on all of its neighbors inside a bucket, i.e., it may be that their assignment is chosen adversarially. As we can guarantee that each event has to give up on at most one bucket, it turns out to suffice to use a constant number of buckets to get a good split, or more generally $O(1/\varepsilon)$ buckets to get $(1+\epsilon)$-approximate split. Generating this bucket assignment is itself a splitting problem (that we term a *q-divide*) requiring the use of LLL, but one with less moving parts and a much simpler analysis in LOCAL. In the CONGEST model, it still requires a novel post-shattering phase.

Post-shattering in CONGEST. We solve the post-shattering phase as a sequence of successive relaxations, one for each disjoint group of clusters of the network decomposition. In effect, we solve a new LLL for each cluster group, with progressively stricter criteria (due to the accumulated discrepancy). Each relaxation is solved by a randomized, rather than a deterministic, algorithm. Namely, we run $O(\log n)$ independent instances of the Moser-Tardos process on the cluster, and since each succeeds with constant probability, we achieve at least one valid solution, w.h.p. This parallel instance technique was introduced by Ghaffari [21] for problems like $\Delta+1$-coloring, a simpler setting where the problem is always solvable on clusters processed *later*, regardless of how the *earlier* clusters are solved.

In our edge-coloring application, we use splitting to whittle down the degree parameter to a manageable size. Once degrees are down to $\text{poly}\log\log n$, we can simulate the known algorithms from the LOCAL model, including derandomization techniques, to solve them also in $\text{poly}\log\log n$ CONGEST rounds.

For our list coloring results, we use our splitting procedure to first reduce the parameter L and T to $\text{poly}\log\log n$ while keeping the initial ratio $L/T \geq (1+\delta)$ almost intact. Then, in additional color pruning steps we amplify the ratio until it is larger than a sufficiently large constant, at which point the problem can be solved efficiently via a known LLL-based method [18].

1.4 Further Related Work

The only known LLL algorithm in the CONGEST model is by Maus and Uitto [39] who provide a $\text{poly}\log\log n$-round algorithm for LLLs with a polynomial criterion and constant dependency degree. In this work, we observe that their runtime remains $\text{poly}\log\log n$, even if the dependency degree is as large as $\text{poly}\log\log n$, see Lemma 21 for details.

Edge coloring. Dubhashi, Grable, and Panconesi [14] gave a distributed algorithm for $(1+\epsilon)\Delta$-edge coloring based on the Rödl nibble method. Their results only apply to large values of Δ. Elkin, Pettie, and Su [15] extended the reach to arbitrary Δ by reduction to distributed LLL, and obtained improved complexity of $O(\log^* \Delta \cdot \lceil \log n/\Delta^{1-o(1)} \rceil)$. Chang et al. [10] improved the complexity to $O(\log_\Delta n + \log^{3+o(1)} \log n)$ for $\varepsilon^{-1} \in O(1)$, and to $O(\log n)$ for $\varepsilon^{-1} \in \tilde{O}(\sqrt{\Delta})$.

There are clear tradeoffs between the number of colors and the time complexity. Computing $(2\Delta-1)$-edge coloring can be achieved in $\text{poly}\log\log n$ rounds [15] (even in CONGEST [29]), and even in $O(\log^* n)$ rounds for $\Delta \geq \log^2 n$ [15, 29]). Chang et al. [10] showed via the round elimination method that computing a $(2\Delta-2)$-edge coloring requires $\Omega(\log_\Delta \log n)$ rounds. A $\text{poly}(\Delta, \log n)$-round algorithm is known for $\Delta+2$-coloring [47] and very recently for $\Delta+1$-coloring [6]. Chang et al. [10] showed that an (possibly randomized) algorithm for $\Delta+1$-coloring that can start with any partial coloring requires $\Omega(\Delta \log n)$ rounds. They also showed that $(1+\log\Delta/\sqrt{\log\Delta})\Delta$-edge coloring can be found in $O(\log n)$ rounds.

Splitting. Ghaffari and Su [28] gave three LOCAL algorithms for splitting the edges of a graph into two parts such that each node has at most $(1+\varepsilon)\Delta/2$ incident edges in each part, rounded up for their randomized result.

Their deterministic algorithms achieve complexity $O(\varepsilon^{-1}\Delta^2\log^5 n)$ when $\Delta \geq c\cdot\varepsilon^{-1}\log n$, and complexity $O(\varepsilon^{-3}\log^7 n)$ when $\Delta \geq c\cdot\varepsilon^{-2}\log n$, where c is a suitable absolute constant. Their randomized algorithm solves the problem for all Δ in $O(\varepsilon^{-2}\Delta^2\log^4 n)$ rounds. These results were later improved by [25] to $O(\varepsilon^{-1-o(1)}\log n)$ rounds for deterministic algorithms and $O(\varepsilon^{-1-o(1)}\log\log n)$ for randomized algorithms, with stronger guarantees on the split. However, it is unclear whether these edge-splitting algorithms can be extended to the CONGEST model, as the algorithms communicate simultaneously over various long paths in the network. The importance of splitting problems for the area was highlighted in [27] and [2]. The latter gave various direct reductions of the maximal independent set problem and coloring problems to splitting problems. In addition, they studied several weak variants of the splitting problem, e.g., splitting into two parts such that each node needs to have at least one neighbors in each part. They show that even these have a $\Omega(\log_\Delta \log n)$ lower bound for randomized algorithms and $\Omega(\log_\Delta n)$ for deterministic algorithms. They also obtain a $\text{poly}\log\log n$-round algorithm for the weak variant in the special case of regular graphs.

1.5 Outline

In Section 2, we define the models, the setup for the Lovász Local Lemma and introduce notation. Section 3 contains our algorithm for the q-divide that does not just serve as a warm-up for our more involved splitting algorithms, but is also used as a subroutine in the latter. Our main splitting algorithm is presented in Section 4 for LOCAL and in Section 5 for CONGEST. In Section 6, we present our splitting applications for edge coloring in LOCAL, and in Appendix A for CONGEST. In Appendix B, we provide more details on our second application, i.e., list coloring. The appendix also contains our results on bipartite splitting (Theorem 4). Further details, and the missing full proofs appear in the full version [30].

2 Models, Lovász Local Lemma, Shattering, and Notation

LOCAL and CONGEST model [36, 43]. In the LOCAL model, a communication network is abstracted as an n-node graph with maximum degree Δ. Nodes communicate in *synchronous rounds*, in each of which, a node can perform arbitrary local computations and send messages of arbitrary size to each of its neighbors. Initially, each node is unaware of the network topology and at the end of the computation a node has to output its own part of the solution, e.g., the colors of its incident edges in an edge coloring problem. The main complexity measure is the number of rounds until each node has produced an output. The CONGEST model is identical, with the additional restriction that messages contain $O(\log n)$ bits.

Distributed Lovász Local Lemma. There are random variables Var and (bad) events $\mathcal{X}$ at the nodes. Each event X depends on a subset $\mathsf{Var}(X)$ of the random variables. Let $p(X)$ denote the probability that event X occurs. As usual, we want to find an assignment to the variables so that none of the events occur. We form the dependency graph $H = (\mathcal{X}, E_H)$ on the events, where two events $X_1, X_2 \in \mathcal{X}$ are adjacent if they depend on a common variable, i.e., if $\mathsf{Var}(X_1) \cap \mathsf{Var}(X_2) \neq \emptyset$.

In a distributed setting, we assume that each variable and each event is associated with some node of the communication graph G. For most LLL algorithms it is essential that the dependency graph can be simulated efficiently in the communication network. In the LOCAL model, one round of communication in H can be simulated in t rounds if the variables $\mathsf{Var}(X)$ upon which the event X depends are within distance t in G of the node where X resides. Let d be the maximum degree of H, while Δ is the max degree of G.

Normally, an LLL is specified in terms of a function f, such that $p(X)f(d) \leq 1$. The original specification of Lovász has $f(d) = e \cdot d$ and ensures the existence of an assignment of the variables such that all bad events are avoided. In the study of distributed LLL algorithms, the functions d^2 [13], $c \cdot d^8$ [18] (both *polynomial criteria*), and 2^d (*exponential criterion*) [7, 9, 8] have appeared in the literature.

▶ **Example 5** (k-vertex splitting with discrepancy $6\Delta/k$ is an LLL)**.** Let each node in the graph pick one of k parts, $V_1, \ldots, V_k$, uniformly at random. Introduce a *bad event* X_v for each node $v \in V$ that holds if the number of neighbors of v within any one part deviates from the expected value by more than $6\Delta/k$, i.e., if $|N(v) \cap V_i| \neq d(v)/k \pm 6\Delta/k$, for some $i \in [k]$. Formally, there is one *variable* for each vertex indicating the part that the vertex joins. As the event X_v shares variables only with the events in its 2-hop neighborhood, the dependency degree of the LLL is $d \leq \Delta^2$. A Chernoff bound shows that $\Pr(X_v) \leq \exp(-\Omega(\Delta)) = \exp(-\Omega(\sqrt{d}))$. Hence, this splitting problem is an LLL with exponential criterion, if Δ is above an absolute constant.

The constant 6 is chosen somewhat arbitrarily in order to make the Chernoff bound-based claim simple. In the following sections, we aim at splittings with discrepancy $(1 + \varepsilon)\Delta/k$.

Shattering. Our algorithms make use of the influential shattering technique[2] in which one first uses a randomized algorithm to set the values of some of the variables such that *unsolved* parts of the graph induce *small connected components*, which are solved in the *post-shattering phase*. The following lemma shows that the remaining components are indeed small.

▶ **Lemma 6** (Lemma 4.1 of [11]). *Consider a randomized procedure that generates a subset* $\mathsf{Bad} \subseteq V$ *of vertices. Suppose that for each* $v \in V$*, we have* $\Pr[v \in \mathsf{Bad}] \leq \Delta^{-3c}$*, and the events* $v \in \mathsf{Bad}$ *and* $u \in \mathsf{Bad}$ *are determined by non-overlapping sets of independent random variables for nodes with distance larger than* $2c$*. Then, w.p.* $1 - n^{-\Omega(c')}$*, each connected component in* $G[\mathsf{Bad}]$ *has size at most* $(c'/c)\Delta^{2c}\log_\Delta n$*.*

The following standard result solves these small components efficiently.

▶ **Lemma 7** ([46]). *There is a deterministic* LOCAL *LLL algorithm with polynomial criterion that runs in* $\operatorname{poly}\log N$ *rounds on instances of size* N*, even with an ID space of size exponential in* N*.*

Proof Sketch. The result follows with the derandomization of the distributed version of Moser-Tardos [41] via the network decomposition by Rozhon and Ghaffari [46], as explained in [46]. Note that the exponential ID space is not an obstacle in the LOCAL model as it can be circumvented by first computing a $\Theta(T)$-distance coloring with $\operatorname{poly} N$ colors, e.g., by using Linial's coloring algorithm [35] if the algorithm runs in T rounds. ◀

Notation and concentration bounds. Given a graph $G = (V, E)$ and a subset $S \subseteq V$ the induced graph $G[S]$ is the graph with vertex set S that contains all edges of E with both endpoints in S. Similarly, for an edge set $F \subseteq E$, the induced graph $G[F]$ is the graph with edge set F that contains all vertices that appear in an edge of F. We denote $[n] = \{0, \ldots, n-1\}$. We use the following standard concentration bounds (see, e.g., [14]).

▶ **Lemma 8** (Chernoff bounds,[14]). *Let* $\{X_i\}_{i=1}^r$ *be a family of independent binary random variables with* $\Pr[X_i = 1] = q_i$*, and let* $X = \sum_{i=1}^r X_i$*. For any* $\delta > 0$*,* $\Pr[|X - \mathbb{E}[X]| \geq \delta\,\mathbb{E}[X]] \leq 2\exp(-\min(\delta, \delta^2)\,\mathbb{E}[X]/3)$*.*

▶ **Corollary 9.** *With* X *of the same form as in Lemma 8,* $\forall \mu, z$ *s.t.* $z \leq \mu$ *and* $\mathbb{E}[X] \leq \mu$*,* $\Pr[|X - \mathbb{E}[X]| \geq z] \leq 2\exp(-z^2/(3\mu))$*.*

3 Warm-Up: Computing q-divides

For an integer $q \geq 1$, a *q-divide* of a graph is a partition of its vertices into q parts ("buckets") $V_1, \ldots, V_q$ such that each vertex has at most $8\Delta/q$ neighbors in each bucket. We show:

▶ **Theorem 10.** *For any* $\Delta \leq \operatorname{poly}\log n$ *and* $q \in [1, (1/6)\Delta/\ln\Delta]$*, there is a* LOCAL *algorithm to compute a* q*-divide in* $\operatorname{poly}\log\log n$*-rounds.*

We use q-divide as a subroutine in our k-splitting algorithm of Theorem 3. Additionally, the techniques to compute a q-divide serve as a warm up for the more involved algorithm for vertex splitting. There are two crucial differences between a (tight) k-splitting and a

[2] The technique has been used extensively for efficient algorithms for various local distributed graph problems and in particular symmetry breaking problems such as sinkless orientation [28], $\Delta + 1$-vertex coloring [11], Δ-coloring [24], Maximal Independent Set [20], Maximal Matching [4], $(2\Delta - 1)$-Edge-Coloring [4], and also for general LLL algorithms on small degree graphs [18].

q-divide: (1) A splitting guarantees both the minimum and maximum number of neighbors of a node inside each part, while a q-divide gives only an upper bound, (2) the upper bounds asked for by a q-divide are loose, i.e., we deviate by a factor 8 from a perfect partition, while a splitting is within a $(1 \pm \varepsilon)$-factor.

A q-divide is guaranteed to exist by LLL when $q \in O(\Delta/\log\Delta)$. A q-divide can also be defined as a $8\Delta/q$-defective $8\Delta/q$-frugal q-coloring, where x-frugal means that each color appears no more than x times in each neighborhood.

▶ Remark 11. For $\Delta/q = \Omega(\log n)$, there is a trivial zero round CONGEST algorithm for q-dividing. Each vertex assigns itself to a bucket uniformly at random ; each vertex has $d(v)/q \leq \Delta/q$ neighbors in each bucket in expectation. By Lemma 8 (Chernoff bound), for each vertex v and $i \in [q]$, $\Pr[|N(v) \cap V_i| > 8\Delta/q] \leq \exp(-\Delta/q) \in n^{-\Omega(1)}$. Therefore, w.h.p., no vertex has more than $8\Delta/q$ neighbors in a bucket.

For smaller Δ/q we give an algorithm based on shattering (that also works for large Δ/q).

The algorithm is parameterized with a threshold parameter $z(v)$ for each vertex v. For Theorem 10 we set $z(v) = 8\Delta/q$ for all nodes. In the full version, we compute slightly different versions of q-divides with different choices of $z(v)$.

Algorithm. **Phase I:** (Pre-shattering) Each vertex picks one of the first $q/2$ buckets u.a.r. Whenever a node has more than $z(v)$ neighbors in a bucket, it deselects these, i.e., these neighbors are removed from the bucket. **Phase II:** (Post-shattering) The post-shattering instance is formed by all nodes that are not assigned to any bucket, together with their neighbors. The objective is to add each unassigned node to one of the last $q/2$ buckets, such that each node has at most $z(v)$ neighbors in each bucket. In Lemma 13, we show that this problem is an LLL instance with a polynomial criterion, and in Lemma 12 that it is induced by connected components of small size. We solve it via Lemma 7 in LOCAL.

▶ **Lemma 12.** *For threshold discrepancy $z(v) = 8\Delta/q$ for all $v \in V$, the connected components participating in the post-shattering phase of the algorithm are of size* $\text{poly}(\Delta) \cdot \log n$*, w.h.p.*

Proof. For $j \in [q/2]$ and node v, let $D_j(v)$ be the number of neighbors of v in bucket j. We have $E[D_j] = 2\Delta/q$ for each of the $q/2$ buckets. By Chernoff (Lemma 8), a node v has an unusually high number of neighbors ($> z(v) = 8\Delta/q = (1+3)2\Delta/q$) in a given bucket w.p. at most $\exp(-2\Delta/q) \leq \Delta^{-12}$, using $q \leq (1/6)\Delta/\ln\Delta$. A node v takes part in the post-shattering phase if one of its neighbors or v itself renounced its choice of bucket, i.e., if a node in its distance-2 neighborhood had too many neighbors in one of the buckets. This occurs w.p. at most $q \cdot \Delta^2 \cdot \Delta^{-12} \leq \Delta^{-9}$, and is fully determined by the random choices of nodes inside the 3-hop ball around v. Hence, by Lemma 6, the graph is shattered into components of size $O(\Delta^6 \log n)$, w.h.p. ◀

▶ **Lemma 13.** *When $z(v) = 8\Delta/q$, for all $v \in V$, the instances formed in the post-shattering phase are LLL problems with criterion $f(d) = (q/2)\exp(-2\sqrt{d}/q)$ and $d \leq \Delta^2$. For $q \leq (1/6)\Delta/\ln\Delta$, the error probability of the LLL is upper bounded by d^{-5}.*

Proof. Consider the following probabilistic process. Each node picks each part in $[q] \setminus [q/2]$ u.a.r., i.e., with probability $p = 2/q$. For $j \in [q] \setminus [q/2]$, let D_j denote the random variable describing the number of neighbors in bucket j. We have $E[D_j] \leq 2\Delta/q$. Let X_v denote the "bad" event that node v has more than $z(v) = 8\Delta/q$ neighbors in one of the $q/2$ buckets. We analyze the LLL formed by the events X_v and their underlying variables.

The event X_v is fully determined by the random choices of direct neighbors of v. Hence, two bad events X_v and X_w are dependent on a shared variable iff v and w are at distance 2 or less, and each bad event shares a variable with at most Δ^2 other events. Therefore,

the dependency graph of the LLL has maximum degree at most $d \leq \Delta^2$. By Chernoff (Lemma 8), X_v occurs w.p. at most $(q/2)\exp(-2\Delta/q)$. Hence, the LLL has criterion $f(d) = (q/2)\exp(-2\sqrt{d}/q)$, which ranges from being polynomial to exponential depending on how small q is compared to $\Delta \geq \sqrt{d}$. In the worst case, the bound $q \leq (1/6)\Delta/\ln\Delta$ implies that $f(d) \leq (\Delta/(12\ln\Delta))\Delta^{-12} \leq d^{-5}$. ◀

Proof of Theorem 10. The problem is solved by the algorithm above. The runtime is $O(1)$ rounds for the pre-shattering phase, and poly log log n rounds for the post-shattering phase via Lemma 7. To apply this lemma we require Lemma 12 that shows that any component in the post-shattering phase has size $\log n \cdot \text{poly}\,\Delta = \text{poly}\log n$, w.h.p., and that Lemma 13 shows that these components form LLLs with a polynomial criterion. ◀

Note that in the special case of $\Delta/q = \Omega(\log n)$ we get the stronger property that, w.h.p., there will no post-shattering phase (see Remark 11).

4 Vertex Splitting in LOCAL

In this section, we prove the following result on vertex splitting.

▶ **Theorem 3.** *There exists a universal constant $c_3 > 0$ s.t.: For any $\varepsilon > 0$, maximum degree $\Delta \leq \text{poly}\log n$, and $k \leq c_3 \cdot (\varepsilon^4\Delta/\ln\Delta)$, there is a distributed randomized* LOCAL *algorithm to compute a k-vertex splitting with discrepancy $\varepsilon\Delta/k$ in $O(1/\varepsilon) + \text{poly}\log\log n$ rounds.*

When Δ is logarithmically larger than k, there is an easy solution.

▶ **Observation 14.** *If $k \leq \varepsilon^2\Delta/(9\ln n)$, the trivial zero round algorithm in which each node picks one of the k parts u.a.r. results in a k-vertex splitting with discrepancy $\varepsilon\Delta/k$, w.h.p.*

Proof. For a node v and class i, let D be the number of neighbors of v that picked class i. Then $\mathbb{E}[D] = d_v/k$. Let $\mu := \Delta/k$, $z = \varepsilon\Delta/k$. By Corollary 9 (Chernoff bound)

$$\Pr[|D - \mathbb{E}[D]| \geq \epsilon\Delta/k] \leq 2\exp(-z^2/(3\mu)) = 2\exp(-\varepsilon^2\Delta/(3k)) \ .$$

This is at most $2n^{-3}$ when $k \leq \varepsilon^2\Delta/(9\ln n)$, so by union bound over all nodes v and classes i we get a k-splitting w.h.p. ◀

4.1 Shattering for ε-Vertex-Splitting in $O(1/\varepsilon)$ Rounds

Due to Observation 14, the most challenging case for a a poly log log n-round algorithm is when $\Delta \leq \text{poly}\log n$ and $\Delta/k = O(\log n)$ holds. Next, we present our algorithm.

FastShattering. Find a q-divide χ for $q = 24/\varepsilon$. To avoid confusion between this partition of the nodes and that of the k-splitting we are computing, let us refer to χ as a *schedule* of the nodes, made of q *slots*, which we denote by $N_1, \ldots, N_q$. Go through the q slots of χ sequentially, and temporarily assign each node in this slot one of the k parts uniformly at random. If a node has received too few or too many neighbors in a part when processing a slot, we retract the last batch of assignments within the neighborhood of that node and freeze those nodes. We also freeze all nodes within distance 3 that are in later slots. All non-frozen nodes (in slot j) keep their assignment permanently. The frozen nodes then get solved in post-shattering (along with all neighbors acting as events, including non-frozen neighbors). For each $j \in [q]$, there is one such post-shattering instance stemming from nodes that were frozen when processing slot j.

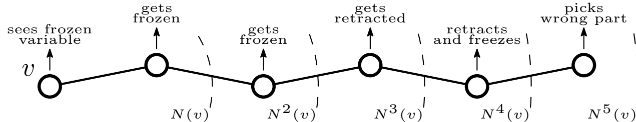


Figure 1 Whether a node joins the post-shattering instance depends on random choices at distance up to 5.

For the rest of this section, set the number of slots to $q = 24/\varepsilon$ and define the following threshold parameters for the pre-shattering and post-shattering phase $z_j^{\mathsf{pre}}(v) = z_j^{\mathsf{post}}(v) = \varepsilon^2\Delta/(72k)$ for all $j \in [q], v \in V$.

Detailed description of FastShattering. During the course of the algorithm nodes are either `frozen` or non-`frozen`. Initially, all nodes are non-`frozen`. Pre-shattering: After computing the q-divide, we iterate through the slots $1, \ldots, q$. In each iteration, we temporarily assign the non-`frozen` nodes in slot j by sampling each u.a.r. into one of the k parts. Next, we formalize the event that retracts these assignments. Fix a node $v \in V$, a slot $j \in [q]$ and a part $i \in [k]$. Let $N_j(v)$ be the neighbors of node v in slot j. Let $\hat{d}_j(v)$ denote the *live degree* of node v when processing slot j, i.e., the number of vertices in $N_j(v)$ that are not `frozen` just before processing slot j. Let $D_{i,j}$ denote the number of neighbors of v in $N_j(v)$ that are temporarily assigned to part i. Event $\mathcal{B}_{i,j}^{\mathsf{pre}}(v)$ holds if $D_{i,j}$ deviates from its expectation $\mathbb{E}[D_{i,j}] = \hat{d}_j(v)/k$ by more than the threshold parameter $z_j^{\mathsf{pre}}(v)$. Let $\mathcal{B}_j^{\mathsf{pre}}(v) = \bigvee_{i \in k} \mathcal{B}_{i,j}^{\mathsf{pre}}(v)$ be the event that v sees such a large deviation from its expectation in some part i. Suppose after sampling event $\mathcal{B}_j^{\mathsf{pre}}(v)$ occurs, then node v undoes the temporal assignment of all neighbors in slot j, i.e., of all nodes in $N_j(v)$, and additionally freezes all unassigned nodes in distance 3, i.e., the nodes in $\{u \in \cup_{j'>j} N_j(v) : d(u,v) \leq 3\}$. Add all nodes that become frozen when processing slot j to Bad_j. All (temporal) assignments that do not undergo a retraction are kept permanently. While `frozen` nodes do not sample colors, each node monitors how its neighbors are being colored and thus yields an *event node* in each of the q iterations, regardless of whether it is `frozen` or not.

Post-shattering: For each $j \in [q]$ there is a separate post-shattering LLL instance containing a variable for each node in Bad_j and a bad event node $\mathcal{B}_j^v$ for each node v with a neighbor in Bad_j. The random process of the j-th LLL is as follows: Each node in Bad_j picks one of the k parts independently and u.a.r. For a node v the number of neighbors in Bad_j is denoted by $f_j(v)$. Let $F_{i,j}(v)$ be the number of neighbors of v in part i (restricted to neighbors in Bad_j). Event $\mathcal{B}_{i,j}^{\mathsf{post}}(v)$ holds if $F_{i,j}$ deviates from its expectation $\mathbb{E}[F_{i,j}(v)] = f_j(v)/k$ by more than the threshold parameter $z_j^{\mathsf{post}}(v)$. The *bad event* $\mathcal{B}_j^{\mathsf{post}}(v) = \bigvee_{i \in k} \mathcal{B}_{i,j}^{\mathsf{post}}(v)$ holds if v sees such a large deviation from its expectation in some part i. In Lemmas 6 and 19 we show that for each $j \in [q]$ we indeed obtain an LLL with polynomial criterion that can be solved via Lemma 7 in the LOCAL model. All q instances are solved in parallel; their deviations add up to $(\varepsilon/3) \cdot \Delta/k$ as shown in Lemma 16.

Intuition for the runtime: The pre-shattering phase runs in $O(q) = O(1/\varepsilon)$ rounds. The post-shattering phase runs in $\text{poly} \log \log n$ rounds for the following reason. Each component in each of the q post-shattering instances forms an LLL and is of size $N = \text{poly}(\Delta) \cdot \log n = \text{poly} \log n$, see Lemma 18. As all components are independent, they can can be solved in parallel in $\text{poly} \log N = \text{poly} \log \log n$ rounds in the LOCAL model via Lemma 7.

Notation. We summarize and extend the notation that we need for the analysis.

- $V_1, \ldots, V_k$ parts (changing throughout the algorithm),
- $N(v)$ neighbors of v in G, $N_j(v)$ neighbors of v in slot j, $\hat{N}_j(v) \subseteq N_j(v)$ are the live neighbors of v in bucket j, i.e., the unfrozen neighbors of v in slot j just before slot j is processed, $F_j(v) \subseteq N(v)$ neighbors of v in the j-th post-shattering instance,
- $d(v) = |N(v)|$, $d_j(v) = |N_j(v)|$, $\hat{d}_j(v) = |\hat{N}_j(v)|$, $f_j(v) = |\hat{F}_j(v)|$.

▶ **Observation 15.** *In FastShattering, any node can have at most one slot in which (some) of its neighbors get their part assignment undone.*

Proof. Let v be a node with a neighbor $u \in N_j(v)$ that has its assignment retracted during slot j. Then u is adjacent to a node that detected that too few or too many of its neighbors were assigned a given part when processing slot j. That node is at distance at most 2 from v, and it freezes the nodes in slots higher than j within distance 3. Therefore, all the unassigned neighbors of v are frozen, and v will not see another retraction in its neighborhood (in fact, it will not even see an assignment). ◀

4.2 Analysis of Discrepancy

In this section, we bound the deviation in the number of neighbors that a node v sees in the i-th part from $d(v)/k$.

The full proof of the following lemma appears in Appendix C.

▶ **Lemma 16.** *In the final assignment $V_1, \ldots, V_k$, i.e., after the pre- and post-shattering phases, each node v has $d(v)/k \pm \varepsilon\Delta/k$ neighbors in every V_i, $i \in [k]$.*

Proof sketch. The discrepancy (deviation from expectation) for a node comes from three sources, (a) slots with neighbors that got retracted, (b) the parts of other slots assigned in the pre-shattering phase, and (c) the deviation summed up over all q post-shattering instances. Due to Observation 15, there can be at most one slot with retracted nodes and the deviation from that slot j^* from the expectation can be upper bounded by $d_{j^*}(v)/k + z \leq 8\Delta/q + z \leq \varepsilon/3\Delta + z$. For each other slot the discrepancy is at most z, and for each of the q instances in the post-shattering phase the discrepancy is also at most z. Thus, with $\sum_{j\in[q]} z_j^{\mathsf{pre}}(v) = \sum_{j\in[q]} z_j^{\mathsf{post}}(v) \leq \varepsilon/3 \cdot \Delta/k$ and $q = 24/\varepsilon$ the total discrepancy adds up to $8\Delta/q + \sum_{j\in[q]} z_j^{\mathsf{pre}}(v) + \sum_{j\in[q]} z_j^{\mathsf{post}}(v) \leq \varepsilon\Delta/k$. ◀

4.3 Analysis of Bad Event Probabilities

Throughout our analysis of the pre-shattering and post-shattering parts of our algorithm, we consider random processes and events which are essentially always the same: nodes in some subgraph each pick a random bucket u.a.r. independently from other nodes, and for each node we analyze the probability that the number of neighbors that pick a given bucket deviates too much from expectation. Recall, that we set $q = 24/\varepsilon$ and $z = \varepsilon^2\Delta/(72k)$ earlier.

▷ Claim 17. Let k, N be positive integers. Let D be a sum of at most N independent Bernouilli random variables of parameter $1/k$, and let $z \leq N/k$. Consider the event $\mathcal{B}$ that D deviates from its expectation by more than z. $\Pr(\mathcal{B}) \leq 2e^{-z^2k/(3N)}$.

In particular, for $N = \Delta$, $k \leq \varepsilon^4\Delta/(2^{19}\ln\Delta)$ and $z = \varepsilon^2\Delta/(72k)$ we obtain $\Pr(\mathcal{B}) \leq \Delta^{-24}$. If D is a sum of only $N = 8\Delta/q$ variables, $k \leq \varepsilon^3\Delta/(2^{17}\ln\Delta)$ suffices for the same bound.

Throughout this paper, D is taken to be a sum of indicator random variables associated to a set of nodes. More precisely, for a subset of nodes in a neighborhood $N(v)$, we consider the sum of the random variable indicating whether each node chose a specific part i out of k choices.

Proof. The general bound on $\Pr(\mathcal{B})$ is from Corollary 9 (Chernoff bound).

When $N = \Delta$, $k \leq \varepsilon^4\Delta/(2^{19}\ln\Delta)$ and $z = \varepsilon^2\Delta/(72k)$, $\exp(-z^2/(3N))$ simplifies to $\exp(-\varepsilon^4\Delta/(3 \cdot 72^2 k)) \leq \Delta^{-24}$. When $N = 8\Delta/q$ (recall $q = 24/\varepsilon$), $k \leq \varepsilon^3\Delta/(2^{17}\ln\Delta)$ and the same z as before, $\exp(-z^2k/(3N))$ simplifies to $\exp(-\varepsilon^3\Delta/(72^2k)) \leq \Delta^{-24}$. ◁

4.4 Analysis of FastShattering

Next, we show that the post-shattering instances consist of small connected components.

▶ **Lemma 18.** *After FastShattering, each connected component in each of the q post-shattering instances is of size $\Delta^{10}\log n$, w.h.p.*

The proof of Lemma 18 appears in Appendix C.2. In spirit it is similar to the proof of Lemma 12, but it is more advanced as frozen variables need to be taken care of formally.

4.5 Post-shattering

In this section we show that the q post-shattering instances are indeed LLLs.

▶ **Lemma 19.** *Each connected component in each of the $q = O(1/\varepsilon)$ post-shattering instances forms an LLL with dependency degree d', bad events' probabilities upper bounded by p' such that the polynomial criterion $d'^8p' < 1$ holds. In LOCAL, the dependency graph can be simulated with $O(1)$ overhead in the communication network G.*

Proof. Consider a post-shattering instance $j \in [q]$. The LLL is formally defined in Section 4.1. Recall, that in the associated random process each node in Bad_j joins one of the k parts u.a.r. and that there is a bad event $\mathcal{B}^{\mathsf{post}}(v) = \bigvee_{i\in[k]}\mathcal{B}_i^{\mathsf{post}}(v)$ for each node with neighbor in Bad_j. The event $\mathcal{B}^{\mathsf{post}}(v)$ occurs if too many or too few neighbors join the i-th part. Thus, a bad event only depends on the randomness of adjacent nodes and the dependency degree is at most $d' = \Delta^2$.

By Claim 17 (applied with $N = \Delta$, and $z = z_j^{\mathsf{post}}(v) = \varepsilon^2\Delta/(72k)$), the probability that $\mathcal{B}_i^{\mathsf{post}}(v)$ holds is at most Δ^{-24} if $k \leq \varepsilon^4\Delta/(2^{19}\log\Delta)$. With a union bound over all k parts we obtain the upper bound $p' = k\Delta^{-24} = \Delta^{-23}$ for the probability of each bad event.

Hence, we obtain $p'd'^{11} < 1$. ◀

4.6 Proof of Theorem 3

Assume $k \leq \varepsilon^4\Delta/(2^{19}\log\Delta)$ and recall that $\Delta \leq \operatorname{poly}\log n$. The runtime of Fast-Shattering is linear in the number of slots, i.e., $O(q) = O(1/\varepsilon)$. Next, we show that the post-shattering instances meet the requirements of Lemma 7. Due to Lemma 18 each connected component is of size $\operatorname{poly}(\Delta)\log n = \operatorname{poly}\log n$, w.h.p. Further, due to Lemma 19 each such component forms an LLL with polynomial criterion and the dependency graph can be simulated with $O(1)$ overhead in the communication network G. Thus, we can apply Lemma 7 (in parallel for all q instances) and obtain a runtime of $\operatorname{poly}\log\log n$ rounds for the post-shattering phase. Lemma 16 shows that the deviation of $|N(v) \cap V_i|$ from $d(v)/k$ is upper bounded by $\varepsilon\Delta/k$ for any node $v \in V$.

5 Vertex Splitting in CONGEST

We obtain the following theorem for vertex splitting and bipartite vertex splitting.

▶ **Theorem 20.** *There exists a universal constant $c_4 > 0$ s.t.: For $\varepsilon > 0$, $\Delta \leq \text{poly}\log n$, and $k \leq c_4 \cdot (\varepsilon^4 \Delta/(\ln \Delta \log^2 \log n))$, there are distributed* CONGEST *algorithms to solve the k-vertex splitting problem with discrepancy $\varepsilon\Delta/k$ and to solve the bipartite k-vertex splitting problem with discrepancy $\varepsilon\Delta_{\mathsf{L}}/k$ in $O(1/\varepsilon) + \text{poly}\log\log n$ rounds.*

Theorem 20 requires k to be a $O(\log^2 \log n)$ factor smaller than in Theorems 3 and 4.

As the pre-shattering phase of Theorems 3 and 4 immediately works in the CONGEST model, the main challenge to prove Theorem 20 is to design a new post-shattering method. Recall, the post-shattering phase in the LOCAL model, i.e, the core steps of Lemma 7. Each connected component in the post-shattering phase forms an LLL with a polynomial criterion and has $N = \text{poly}(\Delta) \cdot \log n = (\text{poly}\log\log n) \cdot \log n$ nodes. This small size allows to compute a network decomposition (see Section 5.1) with $\text{poly}\log\log n$ cluster diameter and $O(\log\log n)$ color classes with distance $s = \Omega(\log N) = \Omega(\log\log n)$ between clusters of the same color. The latter is sufficient to derandomize the $O(\log N) = O(\log\log n) \ll s$ round LLL algorithm from [13]. The details of the derandomization are not important, but it is based on *gathering all information in the cluster and close-by nodes.* In the LOCAL model, this can be done in time that is linear in the cluster diameter, i.e., in $\text{poly}\log\log n$ rounds. One can show that in the CONGEST model all information of a cluster can be encoded with $N \cdot \text{poly}\log\log n$ bits. By using a pipelining argument (see the full version and [39] for details) and that the bandwidth of the CONGEST model is $\Theta(\log n)$ bits, one can aggregate all of this information at a cluster leader in $N \cdot \text{poly}\log\log n/\mathsf{bandwidth} + \mathsf{clusterdiameter}$ rounds, as done in [39]. For $\Delta = \text{poly}\log\log n$, we obtain $N = \log n \cdot \text{poly}\log\log n$ and this method runs in $\text{poly}\log\log n$ rounds. In summary, we obtain the following theorem[3] and the corollary thereafter.

▶ **Lemma 21** ([39]). *There is a randomized* CONGEST *algorithm with bandwidth $= \Theta(\log n)$ for LLL instances of size $N \leq \log n \cdot \text{poly}\log\log n$, dependency degree $d \leq \text{poly}\log\log n$ and error probability $p < d^{-4}$, that runs in $\text{poly}\log\log n$ rounds.*

The algorithm works with an ID space that is exponential in N and is correct w.h.p in n.

▶ **Corollary 22.** *There is a randomized* CONGEST *algorithms for LLL with error probability p, dependency degree d and criterion $p < d^{-8}$ that uses $\text{poly}\log\log n$ rounds, whenever $d \leq \text{poly}\log\log n$. Here, the dependency graph is also the communication network.*

Proof. The shattering framework of [18], w.h.p., reduces to the LLL problem to LLL problems with error probability p', the same dependency degree d and criterion $p' < d^{-4}$ on instances of size $N = \log n \cdot \text{poly}\, d$. These can be solved in $\text{poly}\log\log n$ rounds via Lemma 21. ◀

For $\Delta \gg \text{poly}\log\log n$, any such *gather all information* approach inherently requires significantly larger runtimes. The main ingredient for Theorem 20 is a new method for solving the vertex splitting instances in the post-shattering phase, that can deal with degrees as large as $\Delta = \text{poly}\log\log n$ while using only $\text{poly}\log\log n$ rounds. We prove the following theorem.

[3] The proof of Lemma 21 appears in the full version [30]. It is similar to an CONGEST LLL algorithm in [39] for instances of size $N = O(\log n)$ and the case of $d = O(1)$. In fact, following all dependencies on d (and a slightly increased N) in the proof of [39] yields an algorithm with runtime $\text{poly}(d, \log\log n)$, which yields the desired runtime whenever $d = \text{poly}\log\log n$.

▶ **Lemma 23.** *There exists a universal constant $c_5 > 0$ s.t.: For any $\varepsilon > 0$ and any $k \leq c_5 \cdot (\varepsilon^2\Delta/(\log\Delta\log^2\log n))$, there is a* $\operatorname{poly}\log\log n$*-round randomized* CONGEST *algorithm with bandwidth $= \Theta(\log n)$ that computes a k-vertex splitting with discrepancy $\varepsilon\Delta/k$ on instances of size $N \leq \operatorname{poly}\log n$.*

The algorithm works with an ID space that is exponential in N and is correct w.h.p in n.

The proof of Lemma 23 uses network decompositions that we introduce in Section 5.1, before proving the lemma in Section 5.2. In Section 5.3, we prove Theorem 20.

5.1 Network Decomposition

A *weak distance-s (C, β)-network decomposition with congestion κ* is a partition of the vertex set of a graph into clusters $\mathcal{C}_1, \ldots, \mathcal{C}_p$ of (weak) diameter $\leq \beta$, together with a color from $[C]$ assigned to each cluster such that clusters with the same color are further than s hops apart. Additionally, each cluster has a communication backbone, a Steiner tree of radius $\leq \beta$, and each edge of G is used in at most κ backbones. For additional information on such decompositions we refer the reader to [39, 22]. For the sake of our proofs we only require that such decompositions can be computed efficiently (Theorem 24) and that one can efficiently aggregate information in all clusters of the same color in parallel in time that is essentially proportional to the diameter β (see the full version [30] for the precise statement).

▶ **Theorem 24** ([39]). *For any constant $C > 0$ and $s \in \operatorname{poly}\log\log n$, there is a deterministic* CONGEST *algorithm with bandwidth b that, given a graph G with at most n nodes and unique b-bit IDs from an exponential ID space, computes a weak $(C\log n, O(s/C \cdot \log^3 n))$-network decomposition with cluster distance s and congestion $O(s \cdot \log^2 n)$ in $O(\log^7 n \cdot s^2)$ rounds.*

5.2 Efficient Post-shattering in CONGEST (Proof of Lemma 23)

In order to devise an efficient CONGEST post-shattering algorithm, we decompose each small component into small clusters via the network decomposition algorithm from Theorem 24. Then, the objective is to iterate through the color classes of the decomposition and when processing a cluster we want to assign all nodes in that cluster to a part. When doing so we ensure that each node of the graph obtains a discrepancy of at most $(\varepsilon/Q)\Delta/k$ in each iteration. Hence, over the Q iterations, each node's discrepancy adds up to at most $\varepsilon\Delta/k$.

Proof of Lemma 23. First, compute a distance-3 network decomposition of the graph with $Q = 2\log\log n$ colors via Theorem 24. Then, iterate through the color classes of the network decomposition, processing all clusters of a color class as it gets considered.

When processing a cluster $\mathcal{C}$, we set up a new instance of the vertex splitting problem as follows: Let $V^{\mathsf{L},\mathcal{C}} = N(\mathcal{C})$ be all nodes that have a neighbor in $\mathcal{C}$; $V^{\mathsf{L},\mathcal{C}}$ may contain many nodes of $\mathcal{C}$ itself. Each node of $\mathcal{C}$ is supposed to join one one the parts $V_1^{\mathcal{C}}, \ldots, V_k^{\mathcal{C}}$ such that for each $i \in [k]$ each node in $v \in V^{\mathsf{L},\mathcal{C}}$ has $d_{\mathcal{C}}(v)/k \pm \varepsilon/Q \cdot \Delta/k$ neighbors in $V_i^{\mathcal{C}}$. After processing all clusters we set $V_i = \bigcup_{\text{cluster } \mathcal{C}} V_i^{\mathcal{C}}$. As clusters processed at the same time are in the same color class, they have distance-3, and no node has neighbors in more than one simultaneously processed cluster. Hence, the deviation of the number of neighbors into one V_i from $d(v)/k$ is bounded by $Q \cdot \varepsilon/Q \cdot \Delta/k = \varepsilon\Delta/k$.

The bounds on k and Q imply that the problem that we solve when processing one cluster is an LLL $\mathcal{L}_{\mathcal{C}}$ with a polynomial criterion: Variables and the random process are given by the nodes of $\mathcal{C}$ choosing one of the parts $V_1^{\mathcal{C}}, \ldots, V_k^{\mathcal{C}}$ uniformly at random. For a node $v \in V^{\mathsf{L},\mathcal{C}}$ introduce a bad event $\mathcal{B}_v^{\mathcal{C}}$ that holds if for any $i \in [k]$ node v does not have $d_{\mathcal{C}}(v)/k \pm \varepsilon/Q \cdot \Delta/k$ neighbors in part $V_i^{\mathcal{C}}$. Due to the distance between clusters no node can have a bad event for more than one of the simultaneously processed clusters.

Due to Claim 17 (applied with $N = \Delta$, $z = \varepsilon/Q \cdot \Delta/k$), we obtain that $\Pr(\mathcal{B}_v^{\mathcal{C}}) \leq k \cdot \exp(-\varepsilon^2\Delta/(3Q^2k))$. Plugging in $Q = 2\log\log n$ and $k \leq C\varepsilon^2\Delta/(\log\Delta\log^2\log n)$, we get $\Pr(\mathcal{B}_v^{\mathcal{C}}) \leq k \cdot \exp(-\log\Delta/(12C)) \leq \Delta^{-19}$ for $C \leq 2^{-8}$. As the dependency degree is at most Δ^2, we obtain an LLL with a polynomial criterion of exponent 9.

The goal is to assign all nodes of $\mathcal{C}$ to a part such that all bad events $\mathcal{B}^{\mathcal{C}}(v)$ for $v \in V^{\mathsf{L},\mathcal{C}}$ are avoided. In order to do so, we run $\ell = 6\log n$ parallel instances of the LLL algorithm of [13] on $\mathcal{L}_{\mathcal{C}}$, each running for $O(\log N) = O(\log\log n)$ rounds. At the end of the proof we reason that these ℓ instances can indeed be run efficiently in parallel, for now, we continue with the remaining steps of the algorithm. We say that an instance is *correct* for an event of $\mathcal{L}_{\mathcal{C}}$ if it is avoided under the computed assignment of the instance. By the properties of the algorithm of [13], each instance is correct for all events of $\mathcal{L}_{\mathcal{C}}$ with probability $\geq (1 - 1/N) \geq 1/2$. Hence, with probability $1 - 1/2^{\ell} = 1 - 1/n^6$ one of the ℓ instances is *correct* for all events of $\mathcal{L}_{\mathcal{C}}$. Then, each node holding an event of $\mathcal{L}_{\mathcal{C}}$ determines which instances are correct, and the nodes agree on a *winning* instance, i.e., one that is correct for all of them.

Assume that nodes know in which instance their bad events are avoided. Then, agreeing on a winning instance can be done efficiently as follows: Let each such node hold a bit string of length $\ell = O(\log n)$ in which the j-th bit indicates whether the bad event is avoided in the outcome of the j-th instance. All nodes can agree on a winning instance in time linear in the cluster's weak diameter by computing a bitwise-AND of the bitstrings (see the full version for details).

In order to determine the status of its events in each of the ℓ instances, node v only needs to know which part each neighbor has chosen in which instance. As there are only k parts, the index of the part can be communicated with $O(\log k)$ bits. Hence, a node u can inform each neighbor about the parts node u chose in all ℓ instances by communicating $\ell \cdot O(\log k) = O(\log n \log\log n)$ bits over each incident edge. Using $\mathsf{bandwidth} = \Theta(\log n)$, this requires $O(\log\log n)$ rounds. The same reasoning is also sufficient to run the ℓ instances of [13] in parallel. In one iteration of [13], the variables of local ID minima in the graph induced by violated events are re-sampled. We just reasoned that a node can determine the status of its events in each of the ℓ instances in $O(\log\log n)$ rounds, and with an additional round we can compute a set of local ID minima of violated events for each instance. Then, nodes can inform neighbors about the instances in which they need to re-sample their part. ◀

5.3 Proof of Theorem 20

The pre-shattering phase of computing the q-divide can immediately be implemented in the CONGEST model. Its post-shattering phase is replaced with the stronger q-vertex splitting result of Lemma 23 (with $\varepsilon = 1$ and $k = q = 24/\varepsilon$) that runs in $\operatorname{poly}\log\log n$ rounds. Note that the hypotheses of Theorem 20 assume that $\Delta/(\log\Delta\log^2\log n)$ is greater than an absolute constant $1/c_4$. With $c_4 \leq c_5/24$, q satisfies the hypotheses of Lemma 23.

The pre-shattering of the main algorithm can also immediately be implemented in the CONGEST model. For each of its q post-shattering instances we use Lemma 23 with $\varepsilon^2/72$ and the same k. Using the proof of Lemma 16, the total discrepancy of the pre-shattering and the post-shattering phase is upper bounded by $(2\varepsilon/3)\Delta/k$ and $(\varepsilon/3)\Delta/k$, respectively.

6 Application: $(1+\varepsilon)\Delta$-edge coloring

In this section, we first prove the LOCAL version of the following theorem.

▶ **Theorem 1** (Edge coloring). *For any constant $\varepsilon > 0$, there is a* $\text{poly}\log\log n$*-round randomized* CONGEST *algorithm to compute a $(1+\varepsilon)\Delta$-edge coloring on any graph with maximum degree $\Delta \geq \Delta_0$ where Δ_0 is a sufficiently large constant.*

We use the following result based on prior work to color small degree graphs.

▶ **Theorem 25** ([18, 15, 10]). *For any constant $\varepsilon > 0$, there is an absolute constant Δ_0 such that for $\Delta \geq \Delta_0$, there is a randomized* LOCAL *algorithm with runtime $O(d^2) + \text{poly}\log\log n$ for $(1+\varepsilon)\Delta$-edge coloring where $d = \text{poly}\,\Delta$.*

The papers [15, 10] both solve the $(1+\varepsilon)\Delta$-edge coloring problem via a constant number of LLL iterations (for constant $\varepsilon > 0$). Their dependency graph can be simulated in the original network with $O(1)$ overhead and has dependency degree $d = \text{poly}\,\Delta$. Plugging in the runtime of $O(d^2) + \text{poly}\log\log n$ for solving such LLLs by [18] yields Theorem 25.

High level overview $(1+\varepsilon)\Delta$-edge coloring algorithm. We recursively (two recursion levels) partition the edge set of G into parts that induce small degree subgraphs. Then, we color each subgraph with a disjoint color palette. More detailed, first we partition the edge set into $k = \Theta(\varepsilon^2\Delta/\log n)$ parts such that each part induces a graph of maximum degree at most $\Delta' = \text{poly}\log n$. Then, in another recursive step we partition the edge set of each of these parts further into $k' = \Theta(\varepsilon^4\Delta'/\log^2\log n)$ parts, each with maximum degree $\Delta'' = \text{poly}\log\log n$. We obtain $k \cdot k'$ subgraphs, each with maximum degree at most Δ''. We color each part with a disjoint color palette with $(1+\varepsilon/10)\Delta''$ colors via Theorem 25 in $O((\Delta'')^2) + \text{poly}\log\log n = \text{poly}\log\log n$ rounds. The colors of the $k \cdot k'$ subgraphs sum up to $(1+\varepsilon)\Delta$ colors in total.

Proof of Theorem 1, LOCAL. If $\Delta \leq \text{poly}\log\log n$ we skip the first two steps of the algorithm and immediately apply Theorem 25 to compute a $(1+\varepsilon)\Delta$-edge coloring in $\text{poly}\log\log n$ rounds. If $\Delta \leq \text{poly}\log n$, we skip the first step and set $\Delta' = \Delta$, $k = 1$ and $G_1 = G$, otherwise we first partition the graph into $k = (\varepsilon/6)^2\Delta/(9\log n)$ subgraphs $G_1, \ldots, G_k$, each with maximum degree $\Delta' = \Delta/k + (\varepsilon/6)\cdot\Delta/k = \text{poly}\log n$. To this end, let each edge uniformly at random and independently join one of the G_i's. The same Chernoff bound as in Observation 14 shows that w.h.p., the maximum degree of each G_i is upper bounded by Δ'.

In the next step, we use Theorem 20 to split each $G_i, i \in [k]$ in parallel into $k' = c_4(\varepsilon')^4\Delta'/\log^2\log n$ graphs $G_{i,j}, j \in [k']$, each of maximum degree $\Delta'' = \Delta'/k' + \varepsilon'\Delta'/k' = \text{poly}\log\log n$. We set $\varepsilon' = \varepsilon/6$. Recall, that c_4 is the constant from Theorem 20. More formally, we set up the following k bipartite splitting instances $B_i = (V_i^{\mathsf{L}} \cup V_i^{\mathsf{R}}, E_i)$, $i \in [k]$: $V_i^{\mathsf{R}} = E(G_i)$ and $V_i^{\mathsf{L}} = V(G_i)$. Note that the degree $d^{B_i}(v) = d^{G_i}(v)$ for a node $v \in V_i^{\mathsf{R}}$ and $d^{B_i}(e) = 2$ for a node $e \in V_i^{\mathsf{R}}$. Hence, B_i has maximum degree Δ'.

We use Theorem 4 (for each B_i in parallel and with the same k' and ε') to compute a partition of V_i^{R} into $V_{i,1}^{\mathsf{R}}, \ldots, V_{i,k'}^{\mathsf{R}}$ such that each $v \in V_i^{\mathsf{L}}$ has $d^{B_i}(v)/k' \pm \varepsilon'\Delta'/k' = d^G(v)/(k\cdot k') \pm 3\varepsilon'\Delta/(k\cdot k')$ neighbors in each V_i^{R}. Now, for $i \in [k], j \in [k']$ let $G_{i,j} = G_i[V_{i,j}^{\mathsf{R}}]$ and note that $G_{i,j}$ has maximum degree at most $\Delta'' = \Delta'/k' + \varepsilon'\Delta'/k' = \text{poly}\log\log n$.

In the last step, we apply Theorem 25 on each $G_{i,j}, i \in [k], j \in [k']$ in parallel to edge-color $G_{i,j}$ with $(1+\varepsilon/6)\Delta''$ colors in $\text{poly}\,\Delta'' + \text{poly}\log\log n = \text{poly}\log\log n$ rounds.

The total number of colors used is upper bounded by

$$k \cdot k' \cdot (1+\varepsilon/6)\Delta'' \leq k \cdot (1+\varepsilon/6)^2 \cdot \Delta' \leq (1+\varepsilon/6)^3 \cdot \Delta \leq (1+\varepsilon)\Delta \ . \quad ◀$$

References

1. Noga Alon. A parallel algorithmic version of the local lemma. *Random Structures & Algorithms*, 2(4):367–378, 1991.
2. Philipp Bamberger, Mohsen Ghaffari, Fabian Kuhn, Yannic Maus, and Jara Uitto. On the complexity of distributed splitting problems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 2019. doi:10.1145/3293611.3331630.
3. Leonid Barenboim and Michael Elkin. Distributed $(\Delta+1)$-coloring in linear (in Δ) time. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 111–120, 2009.
4. Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *Journal of the ACM*, 63(3):20:1–20:45, 2016.
5. József Beck. An algorithmic approach to the Lovász local lemma. I. *Random Structures & Algorithms*, 2(4):343–365, 1991.
6. Anton Bernshteyn. A fast distributed algorithm for $\Delta+1$-edge-coloring. *Journal of Combinatorial Theory, Series B*, 152:319–352, 2022.
7. Sebastian Brandt, Orr Fischer, Juho Hirvonen, Barbara Keller, Tuomo Lempiäinen, Joel Rybicki, Jukka Suomela, and Jara Uitto. A lower bound for the distributed Lovász local lemma. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 479–488, 2016.
8. Sebastian Brandt, Christoph Grunau, and Václav Rozhon. Generalizing the sharp threshold phenomenon for the distributed complexity of the Lovász local lemma. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 329–338. ACM, 2020. doi:10.1145/3382734.3405730.
9. Sebastian Brandt, Yannic Maus, and Jara Uitto. A sharp threshold phenomenon for the distributed complexity of the Lovász local lemma. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 389–398. ACM, 2019. doi:10.1145/3293611.3331636.
10. Yi-Jun Chang, Qizheng He, Wenzheng Li, Seth Pettie, and Jara Uitto. Distributed edge coloring and a special case of the constructive Lovász local lemma. *ACM Transactions on Algorithms (TALG)*, 16(1):1–51, 2019.
11. Yi-Jun Chang, Wenzheng Li, and Seth Pettie. Distributed $(\Delta+1)$-coloring via ultrafast graph shattering. *SIAM Journal on Computing*, 49(3):497–539, 2020.
12. Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the LOCAL model. *SIAM Journal on Computing*, 48(1):33–69, 2019.
13. Kai-Min Chung, Seth Pettie, and Hsin-Hao Su. Distributed algorithms for the Lovász local lemma and graph coloring. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 134–143, 2014. doi:10.1145/2611462.2611465.
14. Devdatt P. Dubhashi, David A. Grable, and Alessandro Panconesi. Near-optimal, distributed edge colouring via the nibble method. *Theor. Comput. Sci.*, 203(2):225–251, 1998. doi:10.1016/S0304-3975(98)00022-X.
15. Michael Elkin, Seth Pettie, and Hsin-Hao Su. $(2\Delta-1)$-edge-coloring is much easier than maximal matching in the distributed setting. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 355–370, 2015. doi:10.1137/1.9781611973730.26.
16. Paul Erdös and László Lovász. Problems and Results on 3-chromatic Hypergraphs and some Related Questions. *Colloquia Mathematica Societatis János Bolyai*, pages 609–627, 1974.
17. Manuela Fischer. Improved deterministic distributed matching via rounding. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, volume 91 of *LIPIcs*, pages 17:1–17:15. LZI, 2017. doi:10.4230/LIPIcs.DISC.2017.17.
18. Manuela Fischer and Mohsen Ghaffari. Sublogarithmic distributed algorithms for Lovász local lemma, and the complexity hierarchy. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, volume 91 of *LIPIcs*, pages 18:1–18:16. LZI, 2017. doi:10.4230/LIPIcs.DISC.2017.18.

19 Manuela Fischer, Mohsen Ghaffari, and Fabian Kuhn. Deterministic distributed edge-coloring via hypergraph maximal matching. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 180–191, 2017.

20 Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 270–277, 2016.

21 Mohsen Ghaffari. Distributed maximal independent set using small messages. *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 805–820, 2019.

22 Mohsen Ghaffari, Christoph Grunau, and Václav Rozhoň. Improved deterministic network decomposition. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2021. arXiv:2007.08253.

23 Mohsen Ghaffari, David G. Harris, and Fabian Kuhn. On derandomizing local distributed algorithms. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 662–673, 2018.

24 Mohsen Ghaffari, Juho Hirvonen, Fabian Kuhn, and Yannic Maus. Improved distributed delta-coloring. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 427–436, 2018.

25 Mohsen Ghaffari, Juho Hirvonen, Fabian Kuhn, Yannic Maus, Jukka Suomela, and Jara Uitto. Improved distributed degree splitting and edge coloring. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 19:1–19:15, 2017.

26 Mohsen Ghaffari and Fabian Kuhn. Deterministic distributed vertex coloring: Simpler, faster, and without network decomposition. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 1009–1020, 2021. doi:10.1109/FOCS52979.2021.00101.

27 Mohsen Ghaffari, Fabian Kuhn, and Yannic Maus. On the complexity of local distributed graph problems. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 784–797, 2017.

28 Mohsen Ghaffari and Hsin-Hao Su. Distributed degree splitting, edge coloring, and orientations. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2505–2523, 2017.

29 Magnús M. Halldórsson and Alexandre Nolin. Superfast coloring in CONGEST via efficient color sampling. In *Proceedings of the International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, 2021.

30 Magnús M. Halldórsson, Yannic Maus, and Alexandre Nolin. Fast distributed vertex splitting with applications, 2022. doi:10.48550/ARXIV.2208.08119.

31 David G. Harris. Distributed local approximation algorithms for maximum matching in graphs and hypergraphs. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 700–724, 2019.

32 Penny E. Haxell. A note on vertex list colouring. *Comb. Probab. Comput.*, 10(4):345–347, 2001.

33 Ken-ichi Kawarabayashi and Gregory Schwartzman. Adapting local sequential algorithms to the distributed setting. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, volume 121 of *LIPIcs*, pages 35:1–35:17. LZI, 2018.

34 Fabian Kuhn. Weak graph colorings: distributed algorithms and applications. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architecture (SPAA)*, pages 138–144, 2009.

35 Nathan Linial. Distributive graph algorithms – global solutions from local data. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 331–335, 1987.

36 Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992.

37 Yannic Maus. Distributed graph coloring made easy. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architecture (SPAA)*, pages 362–372. ACM, 2021. doi:10.1145/3409964.3461804.

38 Yannic Maus and Tigran Tonoyan. Local conflict coloring revisited: Linial for lists. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 16:1–16:18, 2020. doi:10.4230/LIPIcs.DISC.2020.16.

39 Yannic Maus and Jara Uitto. Efficient CONGEST algorithms for the Lovász local lemma. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, volume 209 of *LIPIcs*, pages 31:1–31:19. LZI, 2021. doi:10.4230/LIPIcs.DISC.2021.31.

40 Michael Molloy and Bruce Reed. Further algorithmic aspects of the local lemma. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 524–529, 1998.

41 Robin A. Moser and Gábor Tardos. A Constructive Proof of the General Lovász Local Lemma. *J. ACM*, pages 11:1–11:15, 2010.

42 Alessandro Panconesi and Aravind Srinivasan. Randomized distributed edge coloring via an extension of the Chernoff-Hoeffding bounds. *SIAM Journal on Computing*, 26(2):350–368, 1997. doi:10.1137/S0097539793250767.

43 David Peleg. *Distributed Computing: A Locality-Sensitive Approach.* SIAM, 2000.

44 Bruce Reed. The list colouring constants. *Journal of Graph Theory*, 31(2):149–153, 1999. doi:10.1002/(SICI)1097-0118(199906)31:2<149::AID-JGT8>3.0.CO;2-\%23.

45 Bruce Reed and Benny Sudakov. Asymptotically the list colouring constants are 1. *Journal of Combinatorial Theory, Series B*, 86(1):27–37, 2002.

46 Václav Rozhoň and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 350–363, 2020.

47 Hsin-Hao Su and Hoa T. Vu. Towards the locality of Vizing's theorem. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 355–364, 2019.

A Edge coloring in CONGEST (similar to Section 6)

We begin with proving a CONGEST counterpart of Theorem 25 for very low degree graphs.

▶ **Theorem 26.** *For any constant $\varepsilon > 0$, there is an absolute constant Δ_0 such that there is a randomized* CONGEST *algorithm with runtime* $\operatorname{poly}\log\log n$ *for* $(1+\varepsilon)\Delta$*-edge coloring any n-node graph with maximum degree* $\Delta_0 \le \Delta \le \operatorname{poly}\log\log n$.

Proof. Recall from the text after Theorem 25 that the edge-coloring problem can be solved via a constant number of LLL instances that are defined on a dependency graph H such that one round of communication can be simulated in $O(1)$ rounds in LOCAL in the original network [10]. The dependency degree of H is $d = \operatorname{poly}\Delta$. Hence, if $\Delta \le \operatorname{poly}\log\log n$, one round of communication in H can be simulated in $\operatorname{poly}\log\log n$ CONGEST rounds in the communication network. The result follows via Corollary 22. The base case for the algorithm of [10] is a 5Δ-edge coloring step, which can also be solved in $\operatorname{poly}\log\log n$ CONGEST rounds [29]. ◀

Proof of Theorem 1, CONGEST**.** We use the same high level algorithm as in the LOCAL model, that is, we first split into subgraphs of $\Delta' = \operatorname{poly}\log n$ maximum degree, then in to subgraphs of $\Delta'' = \operatorname{poly}\log\log n$ degree, which we then color with disjoint color spaces, with $(1+\varepsilon/6)\Delta''$ colors each. We refer to the LOCAL version for further the details on this reduction. Here, we only explain which parts differ in the CONGEST model. First, each edge is simulated by one of its endpoints. The reduction to $\operatorname{poly}\log n$ degrees works in zero rounds, just as in the LOCAL model.

The most challenging part is the reduction from $\operatorname{poly}\log n$ degrees to $\operatorname{poly}\log\log n$ degrees. The pre-shattering phases (in computing the q-divide and in the main algorithm) immediately work in the CONGEST model. We only need to reason that the post-shattering phases (of

q-divide and the q instances in the main algorithm) can be solved via Lemma 23. To this end, we need to run $\ell = O(\log n)$ instances of [13] in parallel with $\mathsf{bandwidth} = \Theta(\log n)$. Observe that $k \leq \operatorname{poly}\log n$ holds. For each node to be able to evaluate the status of its bad events in all ℓ instances in parallel, we have the node simulating each edge send to the other endpoint the parts it chose in all ℓ instances (one part per instance). As the part index of an edge can be encoded with $O(\log k)$ bits, the indices in the ℓ instances can be communicated with $\ell \cdot O(\log k) = O(\log n \log\log n)$ bits. With the available bandwidth this requires only $O(\log\log n)$ rounds. The other messages needed by the algorithm to simulate the ℓ instances of [13] in parallel, such as whether the edge needs be re-sampled in each instance, similarly never exceed $O(\log\log n)$ rounds.

Once the degrees of the subgraphs are at most $\operatorname{poly}\log\log n$ we use Theorem 26 to color the graphs. Formally, the color space is still large. To really use Theorem 26, vertices color each subgraph with colors in the range from 1 to $\operatorname{poly}\log\log n$, and map their color back to the original color space at the end of the computation. ◀

B Application: List Coloring

▶ **Definition 27.** *In an (L, T)-*list-coloring *instance on a graph $G = (V, E)$, each node $v \in V$ is given a list $L(v)$ of colors of at least L such that for each $c \in L(v)$ there are at most T neighbors u of v with $c \in L(u)$. The parameter T is referred to as the* color degree*. Similarly for $c \in L(v)$, $|\{u \in N(v) \mid c \in L(u)\}|$ is the* color degree *of color c for v.*

Note that one cannot generally solve such the problem via a greedy approach, not even centrally. Still, the objective is to find solutions for arbitrary L and T with a ratio L/T as small as possible. Reed [44] gave a simple LLL argument for the existence of a solution when $L/T \geq \lceil 2e \rceil$. This was improved to $L/T = 2$ by Haxell [32]. Reed and Sudakov [45] then showed that $L/T = 1 + o(1)$ suffices. Reed's famous list coloring conjecture states that $L = T + 2$ colors always suffice [44]. We recall Reed's argument for the existence if $L/T > 2e$. In the distributed setting, there is an $O(\log n)$ round algorithm for $L/T \geq (1 + \delta)$ [13], and a $O(\operatorname{poly}\Delta + \operatorname{poly}\log\log n)$ rounds for $L/T \geq C_0$ for a sufficiently large constant C_0 [18].

LLL formulation (for existence only). Suppose each node picks a color from its list uniformly at random. Define a bad event $\mathcal{B}_{u,v,c}$ for each edge $\{u, v\} \in E$ and each color c if both u and v choose the color c. The probability for such an event is at most $p = 1/|L(v)| \cdot 1/|L(u)| \leq 1/L^2$. The dependency degree of these events is $d = 2L \cdot T$, because it can depend on at most L colors for each of the endpoints of the edge and on T other incident edges for each of these colors. Thus, we obtain the LLL criterion $p \cdot (2L \cdot T) = 2T/L$, and hence for $L/T > 2e$, a the standard criterion $epd < 1$ is satisfied and a solution exists.

Distributed results. Reed's argument leads to an $O(\log^2 n)$-round LOCAL algorithm for any $L/T > 2e$ with the classic Moser-Tardos algorithm [41]. Chang, Pettie and Su [13] gave a algorithm for $L/T = 1 + \delta$, with a quite involved analysis, that runs in time $O(\log^* L \max(1, \log n)/D^{(1-\gamma)})$. Fischer and Ghaffari [18] showed using *color pruning* that there exists some (possibly large) constant C to solve (L, T)-list coloring whenever $L/T \geq C$ in $\operatorname{poly}(\Delta, \log\log n)$ rounds.

Our main theorem combines the effectiveness of [13] with the speed of [18].

▶ **Theorem 2** (List coloring). *There is a $\operatorname{poly}\log\log n$-round randomized LOCAL algorithm for the list coloring problem, for any T and L with $L \geq (1 + \delta)T$, for any $\delta > 0$ and any $\Delta \geq \Delta_0$, for some absolute constant Δ_0.*

Similarly to the edge-coloring problem our high level idea is to first reduce the size of the relevant parameters to $\text{poly}\log\log n$, after which we can solve arbitrary LLLs efficiently on the problem. However, the reduction and the base case (once parameters are of size $\text{poly}\log\log n$) are significantly more involved than in the edge-coloring problem. We summarize the main technical lemma showing that we can efficiently reduce the parameters L and T while keeping the ratio of list size and color degree almost the same.

▶ **Lemma 28** (List color sparsification). *There exists a universal constant $c_6 > 0$ s.t.: For any $\varepsilon > 1/\text{poly}\log\log n$ and $k \leq c_6 \cdot (\varepsilon^4 L/\log L)$, there is a $\text{poly}(\varepsilon^{-1}, \log\log n)$-round algorithm for the following list coloring sparsification problem: Given a (L,T)-list coloring instance with $T < L \leq \text{poly}\log n$ on an n-node graph $G = (V,E)$ the goal is to compute a sublist $L'(v) \subseteq L(v)$ for each node yielding a (L',T')-list coloring instance on the same graph with*

$$L' = L/k \pm \varepsilon L/k, T' \leq T/k + \varepsilon T/k \text{ and } L'/T' \geq (1-\varepsilon)L/T. \tag{1}$$

We obtain the same properties in zero rounds if $L > \text{poly}\log n$ and $k \leq \varepsilon^2\Delta/(9\ln n)$ holds for $\Delta = L \cdot T$.

C Missing Proofs

C.1 Vertex Splitting: Bounding the Discrepancy

In this section, we bound the deviation in the number of neighbors that a node v sees in the i-th part from $d(v)/k$. For a node v let $N_{\mathsf{pre}}(v)$ ($N_{\mathsf{post}}(v)$) be the neighbors of v that are permanently assigned to a part in the pre-shattering (post-shattering) phase. Also, let $d_{\mathsf{pre}}(v) = |N_{\mathsf{pre}}(v)|$ and $d_{\mathsf{post}}(v) = |N_{\mathsf{post}}(v)|$. Recall, the definition of $z_j^{\mathsf{pre}}(v) = z_j^{\mathsf{post}}(v) = \varepsilon^2/(72k)$ and $q = 24/\varepsilon$, which immediately yields the following claim.

▷ Claim 29. We have $\sum_{j\in[q]} z_j^{\mathsf{pre}}(v) = \sum_{j\in[q]} z_j^{\mathsf{post}}(v) \leq \varepsilon/3 \cdot \Delta/k$.

▶ **Lemma 16** (restated with details). *In the final assignment $V_1,\ldots,V_k$, i.e., after the pre-shattering and post-shattering phase, we have the following guarantees on the split for each part $i \in [k]$:*

1. *Node v has $d_{\mathsf{pre}}(v)/k \pm 2\varepsilon/3 \cdot \Delta/k$ neighbors in $V_i \cap N_{\mathsf{pre}}(v)$.*
2. *Node v has $d_{\mathsf{post}}(v)/k \pm \varepsilon/3 \cdot \Delta/k$ neighbors in $V_i \cap N_{\mathsf{post}}(v)$.*

In total, for each $i \in [k]$ any node v has $d(v)/k \pm \varepsilon\Delta/k$ neighbors in V_i.

Proof. We first prove the first claim. The discrepancy (deviation from expectation) for a node v comes from two sources: (a) slots with neighbors that got retracted; (b) other slots. We bound both separately. Consider a vertex v and fix a part $V_i, i \in [k]$. For the rest of the proof let $z_j = z_j^{\mathsf{pre}}(v)$. We partition the vertices in $V_i \cap N_{\mathsf{pre}}(v)$ according to the q slots $N_1(v), \ldots N_q(v)$. Due to Observation 15 for at most one j does $N_j(v)$ contain nodes whose values were retracted. Denote this j (if any) by j^*, otherwise set $j^* = \perp$.

▷ Claim 30. If $j^* \neq \perp$, then $|V_i \cap N_{j^*}(v) \cap N_{\mathsf{pre}}(v)| \leq \hat{d}_{j^*}(v)/k + z_j$.

Proof. If v caused the retraction then, $N_{j^*}(v) \cap N_{\mathsf{pre}}(v) = \emptyset$, as v retracted all assignments of nodes in $N_{j^*}(v)$ and froze the nodes (they will only be assigned in the post-shattering phase). Now consider the case that v did not cause the retraction, i.e., $\mathcal{B}_j^{\mathsf{pre}}$ does not occur, and let X_i be the nodes in part i in the temporal assignment of nodes in slot j before any retractions happened (also before the ones caused by nodes $u \neq v$). Since $\mathcal{B}_j^{\mathsf{pre}}$ does not occur, we have $|X_i \cap N_{j^*}| \in \hat{d}_{j^*}/k \pm z_j$. Some nodes of X_i might get retracted by other nodes $u \neq v$, but we obtain $|V_i \cap N_{j^*}(v) \cap N_{\mathsf{pre}}(v)| \leq |X_i \cap N_{j^*}(v)| \leq \hat{d}_{j^*}(v)/k + z_j$. ◁

▷ Claim 31. For each $j \notin [q] \setminus j^*$, we obtain $|V_i \cap N_{\mathsf{pre}}(v) \cap N_j(v)| = \hat{d}_j(v)/k \pm z_j$.

Proof. Since $j \neq j^*$ there are no retracted variables in $N_j(v)$. If the bound in the claim does not hold, then $\mathcal{B}_j^{\mathsf{pre}}(v)$ would have occurred after the sampling, and v would have retracted, a contradiction. ◁

▷ Claim 32. $\sum_{j\in[q],j\neq j^*} \hat{d}_j(v) \leq d^{\mathsf{pre}}(v) \leq \sum_{j\in[q]} \hat{d}_j(v)$.

In the following we omit the explicit dependence on v, e.g., we write $\hat{d}_j$ instead of $\hat{d}_j(v)$. Using, $\sum_{j\in[q]} z_j(v) \leq \varepsilon\Delta/(3k)$ (Claim 29), $\hat{d}_{j*} \leq d_{j*} \leq 8\Delta/q = \varepsilon\Delta/3$ (from the properties of a q-divide), bounds on $|V_i \cap N_{\mathsf{pre}}(v) \cap N_j(v)|$ (Claims 30 and 31), and Claim 32 we obtain

$$|V_i \cap N_{\mathsf{pre}}(v)| \leq \sum_{j\in[q]} \left(\hat{d}_j/k + z_j\right) \leq (d^{\mathsf{pre}} + \hat{d}_{j^*})/k + \sum_{j\in[q]} z_j \leq d^{\mathsf{pre}}/k + 2\varepsilon\Delta/(3k) \quad \text{and}$$

$$|V_i \cap N_{\mathsf{pre}}(v)| \geq \sum_{j\in[q],j\neq j^*} \left(\hat{d}_j/k - z_j\right) \geq \left(d^{\mathsf{pre}} - \hat{d}_{j^*}\right)/k - \varepsilon\Delta/(3k) \geq d^{\mathsf{pre}}/k - 2\varepsilon\Delta/(3k) \ .$$

For the second part of the claim, fix again some $i \in [k]$ and a node v. There are q separate post-shattering instances. Recall, the set of neighbors of a node participating in the j-th instance is denoted by $F_j(v)$ and $f_j(v) = |F_j(v)|$. The solution to the LLL instance yields

$$|V_i \cap F_j(v) \cap N_{\mathsf{post}}(v)| = f_j(v)/k \pm z_j^{\mathsf{post}}(v). \tag{2}$$

Summing over all q post-shattering instances, using $d_{\mathsf{post}}(v) = \sum_{j\in[q]} f_j(v)$ and using Claim 29 to bound $\sum_{j\in[q]} z_j^{\mathsf{post}}(v) \leq \varepsilon\Delta/3 \leq \varepsilon\Delta/2$ yields the second part of the claim. ◀

C.2 Analysis of FastShattering

▶ **Lemma 18.** *After FastShattering, each connected component in each of the q post-shattering instances is of size $\Delta^{10}\log n$, w.h.p.*

Proof of Lemma 18. Let us focus on one post-shattering instance, instance number j, formed of both the nodes in Bad_j that were frozen while processing slot j and all their incident 'events nodes'. Let us say that a node v *triggers* if one of the events $\mathcal{B}_{i,j}^{\mathsf{pre}}$, $i \in [k]$ occurs, i.e., if $D_{i,j}(v)$ deviates too much from expectation. That a node triggers is entirely determined by the random choices of its neighbors. By Claim 17 (applied with $N = 8\Delta/q$, $z = z_j^{\mathsf{pre}}(v) = \varepsilon^2\Delta/(72k)$, and $D_{i,j}(v)$), the probability that a node triggers is at most Δ^{-24}. A variable is `frozen` if it is within distance 3 of a triggering node. A node v joins the post-shattering instance if it is `frozen` itself or one of its neighbors is `frozen`, which depends on whether nodes within distance 4 of v trigger or not, which itself is entirely determined by the random choices within distance 5 of v. Thus, whether two nodes at distance 10 participate in the j-th post-shattering instance depends on two sets of non-overlapping random variables from the processing of slot j.

By a union bound over the Δ^4 nodes in the 4-hop neighborhood and the k parts, a node participates in the j-th post-shattering instance w.p. at most $k(\Delta^4)\Delta^{-24} \leq \Delta^{-19}$. By Lemma 6, the resulting connected components of the post-shattering instance are all of size $O(\Delta^{10}\log_\Delta n)$, w.h.p. ◀

D Bipartite Vertex Splitting and Beyond

Another classic version is to split only one side of a bipartite graph. Given a bipartite graph $(V^{\mathsf{L}} \cup V^{\mathsf{R}}, E)$ and an parameter k the objective is to split the *variable vertices* V^{R} into k parts $V_1^{\mathsf{R}}, \ldots, V_k^{\mathsf{R}}$ such that the degree of every *event vertex* $u \in V^{\mathsf{L}}$ into each part V_i^{R} does not deviate from $d(u)/k$ by too much. More formally, each event node $u \in V^{\mathsf{L}}$ comes with a parameter $z(u)$ that bounds the deviation. Let Δ_{L} and Δ_{R} be the maximum degree of nodes in V^{L} and V^{R} respectively. With the same analysis as for Theorem 3 (reasons below) we obtain the following theorem for bipartite vertex splitting.

▶ **Theorem 4.** *There exists a universal constant $c_7 > 0$ s.t.: For any $\varepsilon > 0$, maximum degree $\Delta \leq \text{poly} \log n$ and $k \leq c_7 \cdot (\varepsilon^4 \Delta_{\mathsf{L}} / \ln \Delta)$, there is a distributed randomized* LOCAL *algorithm to compute a bipartite k-vertex splitting problem with discrepancy $\varepsilon \Delta_{\mathsf{L}}/k$ in $O(1/\varepsilon) + \text{poly} \log \log n$ rounds.*

The simpler q-divide problem also naturally extends to this more general setup. The objective of a bipartite q-divide is to partition the variable vertices into q parts such that each event node has at most $8\Delta_{\mathsf{L}}/q$ neighbors in each part.

▶ **Theorem 33.** *For any $q \in [1, (1/6)\Delta_{\mathsf{L}}/\ln \Delta]$, there is a* LOCAL *algorithm to compute a bipartite q-divide in* $\text{poly} \log \log n$*-rounds.*

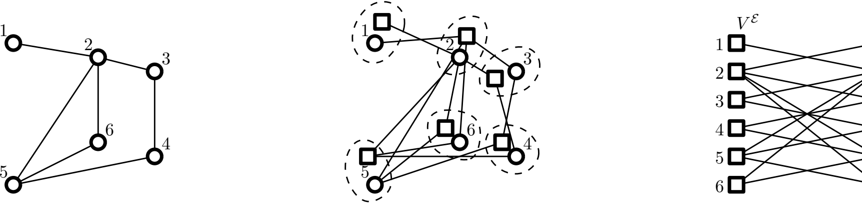


Figure 2 A graph and the bipartite splitting instance obtained from the vertex-splitting problem on it by turning each node into a variable node (circles) and an event node (squares).

To better understand this more general setting and how our results for q-divide and k-split extend to it, let us translate those problems into their bipartite versions. For a graph $G = (V, E)$, we construct a bipartite graph $G' = (V^{\mathsf{L}} \cup V^{\mathsf{R}}, E')$ such that a bipartite k-split (bipartite q-divide) on G' maps to a k-split (q-divide) on G. Let $n = |V|$ be the number of nodes of G and Δ its maximum degree. In bipartite terminology, when computing a k-split on G each node of G is acting both as an event node and a variable node, as we must ensure the proper splitting of its neighborhood as well as assigning it. The bipartite graph corresponding to this problem is the graph $G' = (V^{\mathsf{L}} \cup V^{\mathsf{R}}, E')$ where $|V^{\mathsf{L}}| = |V^{\mathsf{R}}| = n$, $\forall i, j \in |V|^2, v_i v_j \in E \Leftrightarrow \left(v_i^{\mathsf{L}} v_j^{\mathsf{R}} \in E' \wedge v_i^{\mathsf{R}} v_j^{\mathsf{L}} \in E'\right)$. G' has $2n$ nodes and maximum left and right degree $\Delta_{\mathsf{L}} = \Delta_{\mathsf{R}} = \Delta$. See Figure 2 for an illustration of the translation process.

Proof of Theorems 4 and 33. Our proofs of Theorems 3 and 10 naturally extend to the bipartite setting. Intuitively, the algorithms follow the same pattern. We have a simple random assignment procedure that we show properly partitions the neighborhood of a node w.p. $1 - \text{poly}(\Delta)$. In addition, there is a way of running this procedure, retracting some assignments and avoiding to assign some nodes that ensures that only small patches of the graph remain unassigned and the partial assignment that is obtained can be completed to a full assignment. All that we need to show is that our setting of parameters in the bipartite setting are correct, i.e., control the amount of discrepancy as in the previous setting.

Let Δ_{L} be the degree of the left hand side vertices in $V^{\mathcal{E}}$ and let Δ_{R} be the degree of the right hand side vertices in V. Let $\Delta = \max\{\Delta_{\mathsf{L}}, \Delta_{\mathsf{R}}\}$.

All probabilities are exponential in $-\Theta(z)$, or in $\Theta(-z \cdot (z/(d(u)/k)))$ if the degree of a node u is larger than $k \cdot z$.

The union bound in the shattering over distance 5-neighborhoods introduces a multiplicative Δ^5 term. Hence, we require that $e^{\Theta(z)}$ and $e^{\Theta(z)\cdot(z/(d(u)/k))}$ dominate the Δ^5 term. This, clearly holds if $z = \varepsilon^2\Delta/(72k)$ as before, given the assumed upper bound on k.

The proof of the discrepancy (in Section 4.2) remains exactly the same; just note that in the bipartite vertex splitting the discrepancy values ($z(v)$s) depend on Δ_{L} instead of Δ, and hence we obtain a deviation from $d(u)/k$ that is upper bounded by $\varepsilon\Delta_{\mathsf{L}}/k$. ◀

▶ Remark 34. In general, it is not possible to recursively use vertex-splitting to split into smaller and smaller parts. Special properties of an instance (as we have with edge-splitting and when solving list-coloring here) sometime still make it possible.

See the full information for more details on Remark 34

Broadcast CONGEST Algorithms Against Eavesdroppers

Yael Hitron
Weizmann Institute of Science, Rehovot, Israel

Merav Parter
Weizmann Institute of Science, Rehovot, Israel

Eylon Yogev
Bar-Ilan University, Ramat-Gan, Israel

Abstract

An eavesdropper is a passive adversary that aims at extracting private information on the input and output values of the network's participants, by listening to the traffic exchanged over a subset of edges in the graph. We consider secure congest algorithms for the basic broadcast task, in the presence of eavesdropper (edge) adversaries.

For D-diameter n-vertex graphs with edge connectivity $\Theta(f)$, we present f-secure broadcast algorithms that run in $\widetilde{O}(D + \sqrt{fn})$ rounds. These algorithms transmit some broadcast message m^* to all the vertices in the graph, in a way that is information-theoretically secure against an eavesdropper controlling *any* subset of at most f edges in the graph. While our algorithms are heavily based on network coding (secret sharing), we also show that this is essential. For the basic problem of secure unicast we demonstrate a network coding gap of $\Omega(n)$ rounds.

In the presence of vertex adversaries, known as semi-honest, we introduce the *Forbidden-Set Broadcast* problem: In this problem, the vertices of the graph are partitioned into two sets, trusted and untrusted, denoted as $R, F \subseteq V$, respectively, such that $G[R]$ is connected. It is then desired to exchange a secret message m^* between all the trusted vertices while leaking no information to the untrusted set F. Our algorithm works in $\widetilde{O}(D + \sqrt{|R|})$ rounds and its security guarantees hold even when all the untrusted vertices F are controlled by a (centralized) adversary.

2012 ACM Subject Classification Networks → Network algorithms; Theory of computation → Distributed algorithms

Keywords and phrases congest, edge-connectivity, secret sharing

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.27

Funding *Merav Parter*: This project is funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 949083).

1 Introduction

Modern distributed networks are insecure by nature, and consequently, the private information of their users is under a constant threat of leakage to untrusted parties. We consider secure message passing algorithms against eavesdropper adversaries (also known as passive wiretappers) who can eavesdrop on a bounded number of edges in the graph. Our main objective is to provide round-efficient broadcast algorithms that ensure that the eavesdropper obtains no knowledge on the broadcast message, in the information theoretic sense.

The (perfect) secure algorithms in this paper follow the standard congest model [29], where the communication network is abstracted by an n-vertex graph $G = (V, E)$ with unique vertex identifiers of $O(\log n)$ bits. The communication proceeds in synchronous rounds, where in each round, a vertex can exchange $O(\log n)$-bit messages with each of its neighbors. The communication occurs in the presence of a computationally unbounded eavesdropper with full information on the graph topology and the protocols executed by

© Yael Hitron, Merav Parter, and Eylon Yogev;
licensed under Creative Commons License CC-BY 4.0

the vertices. It is oblivious, however, to the internal randomness of the vertices. We strive for *information-theoretic* f-secure algorithms, which informally guarantee that the messages observed by the eavesdropper controlling at most f edges convey no information on the designated transmitted messages. The design of such *round-efficient* secure algorithms finds a wide range of applications, from digital voting systems to cryptocurrencies and blockchain.

While solutions exist under cryptographic assumptions (e.g., using public-key encryption), our main challenge is in providing the graph-theoretical foundations for distributed algorithms with *information-theoretic* (perfect) security. These solutions are usually easier to compute and have everlasting unconditional security, which is not based on computational assumptions that might be broken in the future. Moreover, information-theoretic security is well fitted to the perspective of the congest model. The latter assumes that the vertices are computationally *unbounded*, and therefore it makes sense to assume that the eavesdropper adversary is computationally *unbounded* as well.

Perfect-secure algorithms against eavesdroppers. The notion of perfect security is among the most fundamental and long-studied concepts in cryptography. Within this setting, two main types of passive adversaries have been considered in the literature: semi-honest (vertex adversary) and its edge-analogue, the eavesdropper, which we consider in this paper. Throughout, an algorithm is called f-secure if it provides perfect-security against an eavesdropper controlling f edges in the graph.

There has been a long line of work studying secure algorithms in specific graph topologies and under various model assumptions. Most attention has been devoted for the f-secure unicast problem where it is required for a source s to send a message m^* to a designated target t over an $(f+1)$ edge-connected graph. One of the earliest works in this area is by Dolev, Dwork, Waarts, and Yung [8]. Their f-secure unicast algorithm exchanges the secret message m^* by sending secret-shares of m^* along a collection of $(f+1)$ edge-disjoint s-t paths [1]. In a recent work, Hitron and Parter [15] showed that the length of such edge-disjoint paths might be $\min\{n, (D/f)^{\Theta(f)}\}$. This implies that algorithms that are based on exchanging messages along edge-disjoint paths require linear number of rounds in the worst case, already for $f = \Omega(\log n)$.

The dependency in the length of the edge-disjoint paths appears to be unavoidable at first glance. The reason is that resilience against computationally unbounded adversaries calls for establishing graph-theoretical *secure* channels between pairs of vertices. The natural approach for providing such channels is by means of exchanging information along with a collection of sufficiently many edge (or vertex) disjoint paths; The latter guarantees that some of these paths are fault-free, which allows the endpoints to overcome the adversarial effect. Indeed, so-far, all existing secure broadcast algorithms for general graphs follow the above-mentioned scheme (in one way or the other) and consequently required $\min\{\Theta(n), (D/f)^{\Omega(f)}\}$ rounds [23, 27, 24, 15].

Cai and Yeung [4] provided considerably improved solutions for the secure unicast problem that are based on the notion of *secure network coding* for DAG graphs. Informally, this approach provides the throughput advantage of the standard network coding scheme while also providing perfect resilience against eavesdroppers. Feldman, Malkin, Servedio, and Stein [9] generalized and simplified the method of [4] by designing perfect-secure algorithms to

[1] In their model, the source s and the target t are connected by m wires, and the eavesdropper is listening on a bounded number of wires. These wires may correspond to vertex-disjoint paths or edge-disjoint paths.

exchange a message to a restricted subset of vertices for DAG networks. Jain [16] presented a secure unicast algorithm, which as shown in this paper can be implemented in $O(D)$ congest rounds for D-diameter graphs. A similar algorithm was provided also by Gilboa and Ishai [12]. As secure message transmission algorithms are currently limited to a bounded number of targets and topologies, in this paper we ask the following fundamental question, which for the best of our knowledge, is still vaguely open:

▶ **Question 1.** Is it possible to provide an f-secure broadcast algorithm in sub-linear number of congest rounds, even for $f = \Omega(\log n)$?

A naïve solution for this problem can be provided by simply implementing Jain's [16] algorithm for every target $t \in V$, resulting in $\Omega(n)$ rounds. In this paper, we answer the question above in the affirmative by presenting f-secure broadcast algorithms with round complexity of $\widetilde{O}(D + \sqrt{fn})$. Our algorithms are based on a particular type of network coding, denoted as secret sharing, which is arguably one of the most foundational concepts in cryptography [31]. We combine these tools with the well-known tree-packing of Nash-Williams [22, 5]. We note that while the area of (perfect) secure message transmission against eavesdroppers is well-established in the cryptography and the networking communities, this setting has been barely addressed before in the context of message-passing models with bandwidth limitations, such as the congest model. The only exception is the work of Parter and Yogev [24] that designed f-secure compilers for $f = 1$, that translate any congest (non-secure) r-round algorithm into a 1-secure algorithm with $O(rD)$ rounds. Their (implicit) extension to f faults introduces an overhead of $(fD)^{\Theta(f)}$ rounds.

Network coding gaps in unicast communication. The task of unicast, in which a source vertex s sends information to a designated target t, is one of the most basic and important information dissemination primitives in distributed networks. In typical modern communication networks, it is usually required to run simultaneously many such primitives in parallel. Starting with the influential work of Leighton, Maggs, and Rao [18], the scheduling of multiple unicast tasks has been subject to thorough research over the years, under two main classes of algorithms: (i) *routing-based algorithms* (also known as store-and-forward) where messages are viewed as (atomic) tokens and vertices can only store and forward them, and (ii) *network coding algorithms* where messages are allowed to be mixed together by any form of coding [1]. One of the most intriguing questions in this area is whether coding has a provable advantage over routing-based algorithms. The challenge of providing provable network coding gaps has been addressed extensively over the years [30, 32, 33, 34]. In their influential work, Haeupler, Wajc, and Zuzic [14] demonstrated that the network coding gap for the round-complexity (i.e., makespan) of k unicast tasks is at most polylogarithmic in k. In this work, we aim at understanding the network coding gap for the secure unicast problem. In particular, as all known secure unicast algorithms against eavesdroppers are based on some notion of coding schemes, we ask:

▶ **Question 2.** Is it possible to provide store-and-forward algorithms for secure unicast with sublinear complexity?

We provide a negative answer by establishing a network coding gap of $\Omega(n)$ rounds. To the best of our knowledge, all prior provable gaps were limited to the *logarithmic* regime, even in adversarial settings. For example, Censor-Hillel, Haeupler, Hershkowitz, and Zuzic [6] analyzed the throughput of broadcast algorithms in noisy radio networks, and established a network coding gap of $\Theta(\log n)$. In contrast, for the the fault-free setting, Alon et al. [2] showed that this gap is bounded by a constant.

1.1 Our Results

We present round-efficient secure algorithms for basic communication primitives. Our main result is an f-secure broadcast algorithm for graphs with edge-connectivity of $\Theta(f)$. Throughout, the adversarial edges controlled by the eavesdropper are denoted by $F^* \subseteq E$, and the diameter of the graph is denoted by D. Let $B = O(\log n)$ be the edge bandwidth of the congest model. The *congestion* of a distributed algorithm is an upper bound on the total number of messages that the algorithm sends over a given edge [10].

Warm-Up: Secure unicast. We start by observing that the existing secure network coding protocols for secure-unicast (e.g., by Jain [16]), can be implemented in near-optimal number of congest rounds. Using random scheduling [19, 10], this provides also efficient solutions to multicast tasks[2].

▶ **Lemma 3** (Optimal Secure-Unicast). *Given a D–diameter graph G, there is an $O(D)$-round algorithm that allows a private sender s to send a message m^* to a public target t, while leaking no information to the eavesdropper regarding the message m^* or the identity of the sender s, provided that s and t are connected in $G \setminus F^*$ (i.e., even if $G \setminus F^*$ is not connected).*

A remarkable property of Lemma 3 is that its round complexity is independent of the actual number of faults, and more specifically it does not depend on the diameter of the graph $G \setminus F^*$, as one might expect. Moreover, this unicast algorithm is secure provided that the eavesdropper does not control any s-t cut in the graph (i.e., hence potentially protecting against a large number of adversarial edges).

In the non-secure setting, the scheduling of multiple unicast problems has been usually studied in the store-and-forward model [18, 14]. In this model, the messages are viewed as atomic tokens rather than bits of information, and relay vertices can only forward some of their received messages to their neighbors, but are not allowed to mix messages. While the fault-free setting exhibits at most logarithmic gap w.r.t the number of congest rounds, we show a linear gap for solving a single unicast instance, in the presence of eavesdroppers.

▶ **Lemma 4** (Network Coding Gap for Secure Unicasts). *There exists an n-vertex 2-diameter graph G^*, an s-t pair, and a set F^* of adversarial edges, where s and t are connected in $G \setminus F^*$, that satisfies the following: any store-and-forward algorithm that exchanges a message from s to t in a secure manner must run in $\Omega(n)$ congest rounds (even if the vertices know the topology of the graph). In contrast, using network coding, the problem can be solved in $O(1)$ congest rounds.*

Secure broadcast algorithms. Our main result in this paper is given by an f-secure broadcast algorithm that delivers all the vertices a given (secret) message m^*, while leaking no information to the eavesdropper. We start by providing 1-secure broadcast algorithms for 2 edge-connected graphs. By a direct application of *low-congestion cycle covers* introduced by Parter and Yogev [25, 26], we obtain:

▶ **Theorem 5** (1-Secure Broadcast). *For every 2 edge-connected D-diameter n-vertex graph G, there is a 1-secure randomized broadcast algorithm that runs in $D \cdot n^{o(1)}$ rounds, w.h.p.*

[2] In the multicast problem, a source s sends a message m^* to a subset of targets $U \subseteq V$.

Handling f adversarial edges (rather than just one) using cycle-covers inevitable leads to a round complexity of $(D/f)^{\Theta(f)}$ [15]. We devise a new algorithmic technique for broadcast that overcomes this inherent $(D/f)^{\Theta(f)}$ barrier exhibited by all prior algorithms in the adversarial congest model. In particular, the round complexity of the algorithm is sub-linear for every $f \in o(n)$ faults.

▶ **Theorem 6** (f-Secure Broadcast). *For every $(2f+3)(1+o(1))$ edge-connected n-vertex graph G, there exists a randomized f-secure broadcast algorithm for sending w.h.p a b-bit message m^* that runs in $\widetilde{O}(D+\sqrt{f \cdot b \cdot n}+b)$ rounds. The edge congestion of the algorithm is $\widetilde{O}(\sqrt{f \cdot b \cdot n}+b)$. Moreover, the algorithm can also* hide *the identity of the source vertex holding the message m^*.*

It is interesting to note that our algorithm in fact solves the *anonymous broadcast* problem [20], in which it is also desired to hide the identity of the transmitting source. This problem is used as a black-box in many privacy-preserving applications such as anonymous communication and distributed auctions. Prior work has addressed this problem in all-to-all communication models [21], under cryptographic assumptions [7, 3], or alternatively using a large round complexity [3]. To the best of our knowledge, there has been no round-efficient congest algorithm that works for any sufficiently edge-connected graph topology.

For multiple sources $S \subseteq V$ each holding a distinct b-bit message, one can generalize the above algorithm to show:

▶ **Corollary 7** (f-Secure Multi-Source Broadcast). *Given is a $(2f+3)(1+o(1))$ edge-connected n-vertex graph $G=(V,E)$ and a subset of sources $S \subseteq V$ each holding a b-bit message. There exists a randomized f-secure broadcast algorithm that runs in $\widetilde{O}(D+\sqrt{f \cdot b \cdot |S| \cdot n}+b|S|)$ rounds. The edge congestion of the algorithm is $\widetilde{O}(\sqrt{f \cdot b \cdot |S| \cdot n}+b|S|)$.*

Handling vertex adversaries. A semi-honest adversary controlling a vertex v eavesdrops over the set of all edges incident to v [13]. The broadcast problem as is cannot be defined in the setting of semi-honest adversaries (as by the broadcast definition, all vertices, including the adversarial ones, are required to receive the message). We note however that if the vertices know which of their neighbors are eavesdropped, then it is indeed possible to hide the content of the broadcast message from these vertices, as described next.

We introduce the task of *Forbidden-Set Broadcast* which can be thought of as the *vertex*-analog to broadcast with eavesdroppers. In this broadcast problem the graph $G=(V,E)$ consists of two vertex types: *trusted* receivers R and *untrusted* vertices F, where $R \cup F = V$. It is then desired for the broadcast message m^* to faithfully arrive at all vertices in R, while all untrusted vertices $F \subseteq V$ are required to learn nothing on m^*, in the information-theoretic sense. I.e., the vertices in F are controlled by a semi-honest adversary. This formulation might find many applications in real-life distributed networks, e.g., in settings that call for private information exchange over a distributed network that contains also (untrusted) public data-centers. We show:

▶ **Theorem 8** (Forbidden-Set Broadcast). *Given is a D-diameter graph $G=(V,E)$, a vertex partition into trusted receivers and untrusted vertices $R \cup F = V$, and a source vertex $s \in R$ holding a broadcast message $m^* \in \{0,1\}^B$. There is an $\widetilde{O}(D+\sqrt{|R|})$-round randomized algorithm that allows all vertices in R to receive m^*, w.h.p, while leaking no information to any of the vertices in R, provided that the subgraph $G[R]$ is connected.*

It is easy to see that one can solve this task using $\mathsf{Diam}(G[R])$ rounds. It is also clear that the connectivity of $G[R]$ is a necessary condition. Our improved algorithm is based on sending messages through edges that are incident to the untrusted vertices, while guaranteeing that the semi-honest adversary controlling these edges learns nothing. Using edges in $G \setminus G[R]$ is crucial in order to improve upon the trivial bound of $O(\mathsf{Diam}(G[R]))$ rounds.

A remark on the graph connectivity requirements. It is well-known that f-security requires $(f+1)$ (edge) connectivity, as no security can be provided if the edges controlled by the eavesdropper disconnect the graph. The secure algorithms presented in this paper ask for two types of connectivity requirements. The secure unicast and forbidden-set broadcast procedures of Lemma 3 and Theorem 8 ask for a *minimal* connectivity condition. E.g., as long as s and t are connected in $G \setminus F^*$, the unicast algorithm is secure. In contrast, the broadcast algorithms of Theorems 5 and 6 require that the edge-connectivity of the graph is sufficiently large (above some threshold value). Theorem 5 requires 2 edge-connectivity (which is necessary), while Theorem 6 requires edge-connectivity of $(2f+3)(1+o(1))$. The larger edge connectivity requirement comes from our use of the Nash-William theorem [22], and more specifically from its implementation in the congest model by [5]. Providing efficient broadcast algorithms for $(f+1)$ edge-connected graphs in $o(n)$ rounds is a very interesting open problem.

1.2 Preliminaries

1.2.1 Graph Notation and Basic Distributed Tools

Given a graph $G=(V,E)$, denote its diameter by D. The neighbors of a vertex $v \in V$ are denoted by $N(v)$. Given a tree $T \subseteq G$ and a vertex v, let $\mathrm{ch}(v,T)$ denote the children of v in T, and $\mathrm{par}(v,T)$ is the parent of v. When T is clear from the context, we may omit it. The subtree of T rooted at v is denoted by T_v. For a subset of items X and a probability $p \in [0,1]$, let $X[p]$ denote the random sample obtained by sampling each item of X independently with probability p.

Scheduling of distributed algorithms. Given a graph G, an algorithm $\mathcal{A}$ is said to have congestion at most cong if $\mathcal{A}$ sends at most cong number of messages over each edge in the graph. The dilation of $\mathcal{A}$ is the round complexity of the algorithm. We use the following useful random scheduling procedure due to Leighton, Maggs, and Rao [18], and Ghaffari [10]:

▶ **Theorem 9** ([18, 10]). *Let $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$ be a collection of distributed algorithms such that each algorithm takes at most* dilation *rounds, and there are at most* cong *messages sent through each edge in total throughout the execution of all these algorithms. The algorithms $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$ can be simulated in parallel using $O(\mathrm{cong} + \mathrm{dilation} \cdot \log^2 n)$ rounds w.h.p.*

Basic operations on trees. Given a spanning tree T of diameter $D(T)$, the computation of aggregate functions over a set of input values of $O(\log n)$ bits, can be done in $O(D(T))$ rounds by a simple convergecast algorithm [28].

▷ Claim 10 ([28]). The convergecast of an associative and commutative function g with input values $\{x_u \in \{0,1\}^{O(\log n)}\}_{u \in T_v}$ over a rooted tree T can be performed in $O(D(T))$ rounds and $O(1)$ congestion. Handling k functions $g_1, \ldots, g_k$ can be done in $O(D(T)+k)$ rounds, and congestion $O(k)$.

Our broadcast algorithms are based on decomposing spanning trees into edge-disjoint subtrees (denoted as *fragments*) of bounded size. Since the eavesdropper knows the graph topology, these procedures can be applied in a non-secure manner, as they reveal no information on the secret message(s) m^*. In the full version, we show:

▶ **Lemma 11** (Decomposition of Trees into Small Fragments). *Given an n-vertex spanning tree $T \subseteq G$ and a parameter $K \in [1, n]$, there exists a randomized $\widetilde{O}(K)$-round algorithm* DecomposeTree *for decomposing the edges of T into edge-disjoint sub-trees $T_1, \ldots, T_q \subseteq T$, such that $|T_i| \in \Theta(K)$ for every tree T_i. In the distributed output format, for every sub-tree T_i and a vertex $v \in T_i$, it holds that v knows its incident edges in T_i, as well as, a unique identifier of the tree T_i.*

1.2.2 The Adversarial Setting, Security Definitions and Basic Tools

The adversarial congest model. We consider the standard congest model [29], in the presence of an eavesdropper controlling a fixed set of edges F^*, denoted as *adversarial*. The remaining edges $E \setminus F^*$ are denoted as *reliable*. The vertices do not know the identity of the edges in F^*, but they do know a bound f on their number[3]. The eavesdropper is assumed to be computationally *unbounded*. It is allowed to know the topology of the graph G, and the algorithm description run by the vertices. However, it is oblivious to the internal randomness of the vertices. Our congest algorithms provide a perfect notion of *information theoretic security*, formally defined as follows.

Perfect security against eavesdroppers. For a given randomized algorithm $\mathcal{A}$ running on an n-vertex graph $G = (V, E)$, let $R_e \in \{0,1\}^{**}$ be the random variable specifying all messages sent through e over the execution of $\mathcal{A}$, for every edge $e \in E$. For a subset of edges $F^* = \{e_1, \ldots, e_f\} \subseteq E$, let $M_{F^*} = [R_{e_1}, \ldots, R_{e_f}]$ be the vector of the random variables of the F^* edges. Denote the n-length input (output) vector of the algorithm $\mathcal{A}$ by X (resp., Y).

Algorithm $\mathcal{A}$ is said to be *secure* against an eavesdropper adversary, if for every graph G and every possible assignment of input values x_1, x_2, and output values y_1, y_2, it holds that the following two are equivalent distributions:

$$\{M_{F^*} \mid X = x_1, Y = y_1\} \equiv \{M_{F^*} \mid X = x_2, Y = y_2\} \ .$$

In other words, the random variables M_{F^*} and Y, X are fully independent. A distributed algorithm is f-*secure* if it is secure against an eavesdropper controlling any fixed set of at most f edges.

One-time pad encryption. Our algorithms encrypt messages by XORing them with random keys. This type of encryption is defined as one-time encryption. To prevent the eavesdropper from gaining information on these encrypted messages, it is crucial to use each random key exactly once (hence the phrase one-time). See [17] for further details.

▶ **Definition 12** (One-Time-Pad Encryption). *Let $x \in \{0,1\}^b$ be a b-bit message. In the one-time pad encryption x is encrypted using a uniform random key $K \in \{0,1\}^b$, by setting $\widehat{x} = x \oplus K$. To decrypt $\widehat{x}$ using K, simply let $x = \widehat{x} \oplus K$.*

[3] Note that in the algorithm of Lemma 3 it is not required to know f.

Secret sharing. In a secret sharing scheme, the message is split into multiple parts, called *shares* with the following property: knowing all shares uniquely restores the message, and knowing all but one share reveals no information (in the information theoretic sense) on the original message.

▶ **Definition 13** (Secret Sharing [31]). *Given a bound on the message size B, a message $x \in \{0,1\}^B$ and a parameter k, the secret share* $\mathsf{SecretShare}(x,k,B)$ *is composed of k uniformly randomly chosen strings $x^1,\ldots,x^k \in \{0,1\}^B$ called* shares, *conditioned on $x = \bigoplus_{j=1}^{k} x^j$.*

When the message x represents an integer number bounded by some integer q, the integer-variant, denoted as $\mathsf{SecretShare}_{\text{int}}(m,k,q)$, *splits x into k randomly chosen integer shares $x^1,\ldots,x^k \in \mathbb{Z}_q$ such that $x = (\sum_{j=1}^{k} x^j) \mod q$.*

▶ **Fact 14.** *The collection of k shares obtain by applying* $\mathsf{SecretShare}(x,k,B)$ *(*$\mathsf{SecretShare}_{\text{int}}(m,k,q)$*) satisfies that the joint distribution of any $k-1$ shares is uniformly distributed over $(\{0,1\}^B)^{k-1}$ (resp., $(\mathbb{Z}_q)^{k-1}$).*

2 Secure Unicast

2.1 Unicast and Multicast Algorithms

We start by observing that the existing coding-based algorithms for secure unicast [12, 16] can be implemented in $O(D)$ congest rounds. Recall that in the unicast problem, given a (possible hidden) source s holding a message m^*, it is required for a public target t to receive m^* while leaking no information to the eavesdropping adversary. We next describe a distributed implementation of Jain [16] whose security guarantees hold provided that s and t are connected in $G \setminus F^*$.

High level description of Alg. SecureUnicast(s,t,m^*). The algorithm starts by computing a BFS tree T rooted at the target t, and locally setting a value $x_v = 0$ for every vertex $v \neq s$, and $x_s = m^*$. The message m^* is treated as a field element in $\mathbb{F}_q$ for some sufficiently large polynomial prime q, hence $\sum_v x_v = m^*$. In the first step of the algorithm, the vertices orient the edges of G based on the given tree T. They then secret share their x_v values, and exchange these shares with some of their neighbors (based on the edge-orientation). At this point, each vertex v holds a value y_v which corresponds to the sum of its received share values. The y_v values are then aggregated over the tree T, from the leaf vertices to the root t, which can then compute $\sum_v x_v = m^*$. The security is based on a combinatorial argument which essentially shows that the collection of all messages observed by the eavesdropper is equivalent to a uniform sample of fair coins, hence learning nothing on m^*. The complete proof of Lemma 3 is deferred to the full version.

Simultaneous unicasts and multicast. Using the random delay approach, we can also implement several secure unicast procedures in parallel, and obtain:

▶ **Lemma 15** (Simultaneous Unicasts). *A collection of q instances for the secure unicast problem given by $\{s_i,t_i,m_i\}$ where each $m_i \in \{0,1\}^b$, can be executed in parallel using a randomized algorithm* $\mathsf{SecureSimUnicast}$. *The round complexity is $O(D\log^2 n + q \cdot b/\log n)$ rounds, and the edge congestion is $\widetilde{O}(q \cdot b)$. The security holds provided that every s_i-t_i pair is connected in $G \setminus F^*$. Moreover, the algorithm leaks no information on the identities of the senders $\{s_i\}$, provided that $G \setminus F^*$ is connected.*

The proof of Lemma 15 is deferred to the full version. This immediately yields a secure multicast procedure, where a single source s sends a message m^* to a subset U of vertices. Since our algorithm is based on applying the unicast algorithm with s as the private source, the identity of s can be fully hidden from the eavesdropper.

▶ **Corollary 16** (Multicast). *Given is a private source s, a public subset of vertices $U \subseteq V$ and a b-bit message m^* held by s. There is a randomized algorithm* SecureMulticast(s, U, m^*) *that lets s securely send m^* to all vertices in U provided that s-u are connected in $G \setminus F^*$ for every $u \in U$. The round complexity is $\widetilde{O}(D + b \cdot |U|)$. Moreover, the algorithm leaks no information on the identity of s, provided that $G \setminus F^*$ is connected.*

2.2 A Network Coding Gap of $\Omega(n)$ Rounds for Secure Unicast

In this section, we show that when restricting to the class of store-and-forward algorithms, the task of secure unicast requires $\Omega(n)$ rounds, providing the proof of Lemma 4. Interestingly, we show that this lower bound holds even if the vertices know the identity of the edges in F^*, and the graph topology. In the store-and-forward model, messages are viewed as (atomic) tokens and vertices can only store and forward them. Specifically, in the unicast problem, the source vertex s can send one message to each neighbor in each round, and every other vertex can send in round r only one of the messages it received by round $r-1$ to each neighbor. For every parameter $n \geq 4$, we consider an n-vertex graph G^* defined as follows.

The lower-bound graph G^*. The graph is composed of an $(n-1)$-length s-t path $P = \{v_0 = s, v_1 \ldots, v_{n-2} = t\}$, and an additional vertex u connected to all the vertices in P in a star-shape manner (see Figure 1). The eavesdropper listens to all the edges that are adjacent to u, denoted as F^* (shown in red in the figure). We note that although s and t are connected in $G^* \setminus F^*$, the diameter of $G^* \setminus F^*$ is $\Omega(n)$, while the diameter of G^* is 2. As we will see, our lower bound argument shows that any store-and-forward secure algorithm must run in $\Omega(n)$ rounds on G^*, which stands in stark contrast to the network-coding based solution of Lemma 3 that solves the task in $O(1)$ rounds.

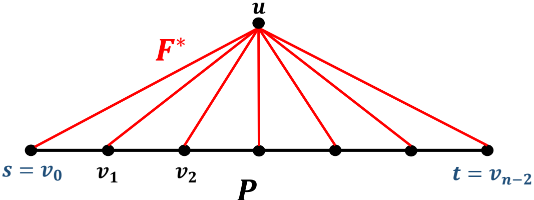


Figure 1 An illustration of the lower bound graph G^*.

On a high level, the key limitation of store-and-forward algorithms that we exploit in our lower bound argument is that all messages sent throughout the algorithm must be originated at the source vertex s. Therefore, for every vertex $v_i \in P$, all messages received by v_i in round $j \leq i-1$ must have been sent over "shortcut" edges which are in F^*. To be more concrete, we show by induction on k, that for every round $k \geq 1$ and $i \geq k+1$, the eavesdropper can calculate the probability distribution of the messages received by v_i by round k. The key inductive argument is that for every vertex v_i and a neighbor $v_j \in N(v_i)$, given the probability distribution on the messages that v_j received up to round $k-1$, as the eavesdropper knows the protocol executed by v_j, it can deduce the probability distribution over the messages v_j sent in the next round k.

It then follows that if the round complexity of the algorithm is at most $n-3$, the eavesdropper can deduce the probability distribution, over all messages received by the target t. Consequently, it can calculate the probability that t outputs a given message. By the correctness of the algorithm, w.h.p, t outputs the correct message m^*, and therefore the eavesdropper learns this message, w.h.p. We next provide the formal lower-bound proof.

Proof of Lemma 4. Let $\mathcal{A}$ be a randomized store-and-forward unicast algorithm, and consider an application of $\mathcal{A}$ on G^* where s wishes to send a message m^* to t. Since $\mathcal{A}$ is *randomized*, in each round k, every vertex can produce a set of random coins. For a vertex $v \in V$ and round k, let $C_{\leq k}(v)$ be the collection of random coins produced by v up to round k, and let $\mathcal{C}_{\leq k} = \bigcup_{v \in V} C_{\leq k}(v)$.

We start by noting that for every vertex v and round r, the messages sent by v in round r are fully determined by the random coins $C_{\leq r}(v)$ and the information that v received by round $r-1$: the messages, along with the rounds and neighbors on which they were received. For each vertex v, we denote the information received by v up to round k by:

$$\mathcal{T}_k(v) = \{(m, r_m, v_m) \mid r_m \leq k,\ v \text{ received the message } m \text{ in round } r_m \text{ from } v_m\}.$$

For every round k, let $M_{F^*}(k)$ be the messages sent over the edges in F^* up to round k. Towards proving Lemma 4, we show that for every round $k \geq 1$ and $i \geq k+1$, the eavesdropper can deduce the following probability distribution on the value of $\mathcal{T}_k(v_i)$:

$$\mathcal{D}(k, v_i) = \{\Pr_{\mathcal{C}_{\leq k}}[\mathcal{T}_k(v_i) = \tau \mid M_{F^*}(k)]\}_{\tau \in \{0,1\}^*}.$$

In Appendix A, we prove by induction the following claim:

▷ Claim 17. For every round $k \geq 1$ and $i \geq k+1$, the eavesdropper can calculate the probability distribution $\mathcal{D}(k, v_i)$ by the end of round k.

We next show that in order to satisfy the security guarantee, the round complexity of Alg. $\mathcal{A}$ must be at least $n-2$. This completes the proof of Lemma 4.

▷ Claim 18. The round complexity of Alg. $\mathcal{A}$ is at least $n-2$.

Proof. Assume by contradiction that the round complexity of $\mathcal{A}$ is bounded by $\mu \leq n-3$. We next show that the eavesdropper can deduce the output message m^*, with high probability, in contradiction to the security guarantee.

Let M_t be the message output by the target vertex t by the end of the algorithm. By the correctness of Alg. $\mathcal{A}$, it holds that:

$$\Pr_{\mathcal{C}_{\leq \mu}}[M_t = m^*] \geq 1 - 1/n^c. \tag{1}$$

Since the algorithm maintains perfect security, the random variables $M_{F^*}(\mu)$ and M_t are fully independent (see Section 1.2.2), and therefore:

$$\Pr_{\mathcal{C}_{\leq \mu}}[M_t = m^* \mid M_{F^*}(\mu)] = \Pr_{\mathcal{C}_{\leq \mu}}[M_t = m^*]. \tag{2}$$

Hence, by Equations (1) and (2) it is sufficient to show that the eavesdropper can compute the probability $\Pr_{\mathcal{C}_{\leq \mu}}[M_t = m^* \mid M_{F^*}(\mu)]$. By the law of total probability, it holds that:

$$\Pr_{\mathcal{C}_{\leq\mu}}[M_t = m^* \mid M_{F^*}(\mu)] =$$
$$= \sum_{\tau\in\{0,1\}^*} \Pr_{\mathcal{C}_{\leq\mu}}[M_t = m^* \mid \mathcal{T}_\mu(t) = \tau, M_{F^*}(\mu)] \cdot \Pr_{\mathcal{C}_{\leq\mu}}[\mathcal{T}_\mu(t) = \tau \mid M_{F^*}(\mu)] \ . \quad (3)$$

We next show that the eavesdropper can compute each term in the right-hand side of Equation (3). We will show that for every value $\tau \in \{0,1\}^*$ the eavesdropper can calculate the probabilities:

$$P_1(\tau) = \Pr_{\mathcal{C}_{\leq\mu}}[M_t = m^* \mid \mathcal{T}_\mu(t) = \tau, M_{F^*}(\mu)], \text{ and } P_2(\tau) = \Pr_{\mathcal{C}_{\leq\mu}}[\mathcal{T}_\mu(t) = \tau \mid M_{F^*}(\mu)].$$

Since $\mu \leq n-3$ and $t = v_{n-2}$, by Claim 17 and the definition of $D(\mu,t)$, for every $\tau \in \{0,1\}^*$ the eavesdropper can calculate $P_2(\tau)$. We next consider $P_1(\tau)$. The output message M_t is fully determined by the tuples in $\mathcal{T}_\mu(t)$ and the random coins $C_{\leq\mu}(t)$. Hence, given the tuples in $\mathcal{T}_\mu(t)$ and the random coins $C_{\leq\mu}(t)$, the eavesdropper can determine the output message M_t (with probability 1). It follows that the eavesdropper can compute the probability:

$$\Pr_{\mathcal{C}_{\leq\mu}}[M_t = m^* \mid \mathcal{T}_\mu(t) = \tau, M_{F^*}(\mu)] = P_1(\tau).$$

Altogether, by combining Equations (1)–(3), it follows that the eavesdropper can deduce the message m^* w.h.p., in contradiction to the security guarantee. ◁

3 Secure Broadcast Algorithms

In this section, we present broadcast algorithms which remain confidential in the presence of an eavesdropping adversary. We start by describing an $\widetilde{O}(D)$-round algorithm for two edge-connected graphs, in the presence of a single adversarial edge, i.e., $|F^*| = 1$. Extending this algorithm to multiple edges f can be provably shown to require $(D/f)^{\Omega(f)}$ rounds (by [15]). We then present an alternative approach that bypasses the $(D/f)^{\Omega(f)}$-barrier at the cost of requiring a slightly larger edge-connectivity. Throughout this section, the source vertex is denoted by s, and the broadcast message by m^*.

3.1 Handling a Single Adversarial Edge

The broadcast algorithm is based on the notion of low-congestion cycle covers introduced by Parter and Yogev [25]. For a given 2 edge-connected graph $G = (V,E)$, a (c,d)-cycle cover is a collection of cycles $\mathcal{C}$ such that (i) each edge $e \in G$ participates in at least one cycle, and at most c cycles in $\mathcal{C}$, and (ii) all cycles are of length at most d.

▶ **Lemma 19** ([25, 26]). *There is a deterministic r-round algorithm that for every n-vertex two edge-connected graph $G = (V,E)$ computes a (c,d)-cycle cover $\mathcal{C}$ for G, where $c = 2^{O(\sqrt{\log n})}$ and $d, r = D \cdot 2^{O(\sqrt{\log n})}$. In the distributed output format, each vertex knows its incident edges on each cycle $\mathcal{C}$ as well as, a unique identifier for these cycles.*

The algorithm. The algorithm has two main phases. The first phase provides a secret key $r_{u,v} \in \{0,1\}^B$ for every neighboring pair u,v in G. These keys are *hidden* from the eavesdropper. The second phase implements a standard (fault-free) broadcast algorithm with the only distinction that the messages exchanged (in the fault-free algorithm) are now encrypted with the $\{r_{u,v}\}$ keys.

The algorithm starts by computing a cycle-cover $\mathcal{C}$ for G using Lemma 19. We use these cycles to provide u, v with a key $r_{u,v}$ that is hidden from the eavesdropper. This is done by letting u generate two independent random keys $r'_{u,v}, r''_{u,v} \in \{0,1\}^B$ which are then sent to v using of the cycle C_e as follows: $r'_{u,v}$ is sent directly over the edge (u,v), and $r''_{u,v}$ is sent to v over the u-v path $C_e \setminus \{(u,v)\}$. The vertex v then sets $r_{u,v} = r'_{u,v} \oplus r''_{u,v}$.

Finally, the algorithm propagates the message m^* from s downwards a BFS tree rooted at s. By round $i \geq 0$ every vertex u at layer i are assumed to know m^*. In round i every vertex u at layer i sends the encrypted message $m^* \oplus r_{u,v}$ to each child v in the tree. This completes the description of the algorithm.

Correctness and security. Each neighboring pair (u,v) share the key $r_{u,v}$. Therefore, each vertex v can decrypt the received message and obtain m^*. Since the eavesdropper controls a fixed edge in the graph, it learns nothing on the collection of $\{r_{u,v} : (u,v) \in T\}$ keys. To see this, consider a single key $r_{u,v}$ exchanged over the cycle C. Since the eavesdropper controls at most one edge on that cycle, it knows either $r'_{u,v}$ or $r''_{u,v}$ (but not both). Therefore it learns nothing on $r_{u,v}$. Since all messages sent through the tree T are encrypted using one-time pad encryption (Definition 12) with their corresponding keys, the eavesdropper learns nothing on the message m^*.

Running time. The time consuming step is the exchange of the keys over all cycles in $\mathcal{C}$. For every edge (u,v), let $\mathcal{A}_{(u,v)}$ be the algorithm that exchanges the $O(\log n)$-length keys $r'_{u,v}, r''_{u,v}$ over the cycle covering (u,v). By the properties of the (c,d) cycle-cover, each cycle $C \in \mathcal{C}$ can be used to cover at most $|C| \leq d$ many edges, and since each edge participates in at most c cycles, overall each edge participates in $O(c \cdot d)$ algorithms. Therefore, the total congestion of the collection of m algorithms is bounded by $O(c \cdot d)$, and the dilation of each algorithm is d. Using the random-delay based scheduling of Theorem 9, we can implement the algorithms $\{\mathcal{A}_{(u,v)} : (u,v) \in E\}$ simultaneously within $\widetilde{O}(c \cdot d)$ rounds. Theorem 5 follows by Lemma 19.

3.2 Handling Multiple Adversarial Edges

We now turn to consider f-secure broadcast algorithms against f adversarial edges F^*, and prove Theorem 6. Let m^* be a b-bit message held by the source. Our f-secure algorithm is based on the distributed computation of a tree collection denoted as *fractional tree packing* [5]. Note that to this date, there is no round-efficient algorithm for computing an *integral* tree-packing.

▶ **Definition 20** (Fractional Tree Packing). *A* fractional tree-packing *of a graph G is a collection of spanning trees $\mathcal{T}$, and a weight function $w : \mathcal{T} \to (0,1]$, such that for every edge $e \in E$, $\sum_{T_i \in \mathcal{T} : e \in T_i} w(T_i) \leq 1$. The size of the fractional tree packing $\mathcal{T}$ is denoted by $\chi(\mathcal{T}) = \sum_{T_i \in \mathcal{T}} w(T_i)$.*

Our algorithm is based on the distributed computation of fractional tree packing, due to Censor-Hillel, Ghaffari, and Kuhn [5]. Our setting requires a slight adaptation of the construction by [5][4], as summarized in the next lemma. See the full version for the proof.

[4] E.g., for our purposes, it is important that the running time would depend on the number of faulty edges, f, rather than on the actual edge connectivity λ, which might be significantly larger than f.

▶ **Lemma 21** (A slight adaptation of Theorem 1.3 [5]). *Given a D-diameter λ edge-connected n-vertex graph, and an integer parameter $\lambda' \leq \frac{\lambda-1}{2}(1-o(1))$, one can compute a fractional tree packing $\mathcal{T}$ and a weight function $w : \mathcal{T} \to (0,1]$, such that:*
1. *for every $T_i \in \mathcal{T}$, $w(T_i) = \frac{n_i}{\lceil \log^8 n \rceil}$ for some positive integer $n_i \geq 1$,*
2. *$\chi(\mathcal{T}) \in [\lambda', \lambda'(1+o(1))]$,*
3. *the round complexity of the algorithm is $\widetilde{O}(D + \sqrt{\lambda' \cdot n})$ and the edge-congestion is $\widetilde{O}(\sqrt{\lambda' \cdot n})$.*

We are now ready to provide the complete description of Alg. SecureEavesBroadcast given a source s holding a broadcast message m^* of b-bits (where possibly $b = \Omega(\log n)$).

Step (0): Fractional tree decomposition. Apply the fractional tree-packing algorithm of Lemma 21 with $\lambda' = f + 1$. Denote the output tree packing by $\mathcal{T}$ and its size by $\chi(\mathcal{T})$. By the end of this computation, the vertices know the weights $n_1, \ldots, n_{|\mathcal{T}|}$ of the trees in $\mathcal{T}$ (see Lemma 21(1)).

Step (1): Multicasting secret shares of m^* to sampled landmarks. The source vertex s secret shares its b-bit broadcast message m^* into

$$\widehat{f} = \chi(\mathcal{T}) \cdot \lceil \log^8 n \rceil \text{ shares} \tag{4}$$

using Procedure SecretShare$(m^*, \widehat{f}, b)$ (Definition 13), denoted as $M^* = (m_1, \ldots, m_{\widehat{f}})$, where each share m_i has b bits. Note that by Lemma 21(1), $\widehat{f}$ is an integer. Next, the algorithm samples a collection of landmarks $L = V[p]$ for

$$p = \Theta(\log n / \min\{\sqrt{f \cdot b \cdot n}, n\}).$$

This is done by letting each vertex join L independently with probability p. The identities of the sampled vertices L are broadcast to all the vertices[5] in $O(D + |L|)$ rounds. Next, the algorithm applies Alg. SecureMulticast(s, L, M^*) of Cor. 16 to securely send the landmarks L the collection of all shares.

Step (2): Fragmentation of tree-packing and leader selections. Decompose each tree $T_i \in \mathcal{T}$ into fragments $T_{i,j}$ of size $\Theta(\min\{\sqrt{f \cdot b \cdot n}, n\})$ using Alg. DecomposeTree(T_i) of Lemma 11. In addition, for every fragment $T_{i,j}$, let $\ell_{i,j}$ be some chosen vertex in $L \cap V(T_{i,j})$, denoted as the *leader*, which exists with high probability. This leader can be chosen in $D(T_{i,j})$ rounds by a simple convergecast over each fragment, simultaneously.

Step (3): Shares propagation in each fragment. Recall that by Lemma 21(1), $w(T_i) = \frac{n_i}{\lceil \log^8 n \rceil}$ for some positive integer $n_i \geq 1$ for every tree $T_i \in \mathcal{T}$. Our goal is to propagate a distinct collection of n_i shares in M^* over each tree T_i. To do that, the landmark vertices partition locally and canonically the shares of M^* into disjoint subsets $M_1^*, \ldots, M_k^*$ such that $|M_i^*| = n_i$ for every $i \in \{1, \ldots, k = |\mathcal{T}|\}$. Note that by Equation (4), $\sum_{T_i \in \mathcal{T}} n_i = \widehat{f}$. The leader $\ell_{i,j}$ of each fragment $T_{i,j}$ sends all the shares in M_i^* over its fragment. Finally, each vertex v recovers the b-bit broadcast message, by setting $m^* = \oplus_{i=1}^{k} \oplus_{m \in M_i^*} m$. This completes the description of the algorithm.

[5] This can be done in a non-secure manner, as the eavesdropper is allowed to know L.

Correctness. Since the edge-connectivity of G is at least $(2f+3)(1+o(1))$, one can obtain a fractional tree packing $\mathcal{T}$ of size $\chi(\mathcal{T}) \in [f+1,(f+1)(1+o(1))]$ using Lemma 21(2). We claim that each vertex receives w.h.p. all the shares in M^* and therefore recovers m^* successfully. Since all the trees $T_i \in \mathcal{T}$ are spanning, a vertex v belongs to some fragment T_{i,j_v} for every $T_i \in \mathcal{T}$. Since each fragment has $|V(T_{i,j})| \in \Theta(\min\{\sqrt{f \cdot b \cdot n}, n\})$ vertices, and each vertex is sampled into L with probability $p = \Theta(\log n/\min\{\sqrt{f \cdot b \cdot n}, n\})$, by the Chernoff bound, w.h.p., each fragment contains some landmark vertex in L. Therefore, each vertex v receives all shares in M_i^* over the fragment T_{i,j_v}, for every $T_i \in \mathcal{T}$. The correctness follows as $\bigcup_{i=1}^k M_i^* = M^*$, where k denotes the number of trees in $\mathcal{T}$.

Security. Recall that the eavesdropper is assumed to know G, therefore Step (0) and Step (2) can be implemented in a non-secure manner. Since $G \setminus F^*$ is connected, Step (1) is f-secure by Cor. 16. We turn to consider Step (3) in which the shares in M^* are sent over the tree fragments of $\mathcal{T}$. We show that there exists at least one share in M^* that the eavesdropper did not learn, and therefore, by Fact 14, it knows nothing on m^*. Since the tree fragments of each $T_i \in \mathcal{T}$ are edge-disjoint, the number of shares in M^* sent over an edge e is bounded by:

$$\sum_{e \in T_i} |M_i^*| = \sum_{e \in T_i} n_i \leq \lceil \log^8 n \rceil \ , \tag{5}$$

where the last inequality follows by the fractional tree packing guarantee that $\sum_{e \in T_i} w(T_i) \leq 1$, and using the fact that $w(T_i) = \frac{n_i}{\lceil \log^8 n \rceil}$ for every $T_i \in \mathcal{T}$. Therefore, the number of shares observed by the eavesdropper can be bounded by:

$$f \cdot \lceil \log^8 n \rceil < \chi(\mathcal{T}) \cdot \lceil \log^8 n \rceil = \widehat{f} = |M^*| \ ,$$

where the first inequality is by Lemma 21(2) (with $\lambda' = f+1$), and the last equality follows by Equation (4).

Round complexity. Finally, we turn to bound the round complexity and the edge congestion of the algorithm. In Step (0) the computation of the fractional tree packing can be done in $\widetilde{O}(D + \sqrt{f \cdot n})$ rounds by Lemma 21(3) with $\lambda' = f+1$. Step (1) takes $\widetilde{O}(D + f \cdot b \cdot |L|)$ rounds, by Cor. 16. Since, w.h.p., $|L| = \widetilde{O}(\sqrt{n/(f \cdot b)})$, it takes $\widetilde{O}(D + \sqrt{f \cdot b \cdot n})$ rounds. By Lemma 11 Step (2) takes $\widetilde{O}(\min\{\sqrt{f \cdot b \cdot n}, n\})$ rounds. As for Step (3), since $w(T_i) \leq 1$, it holds that $n_i \leq \lceil \log^8 n \rceil$ for every $T_i \in \mathcal{T}$. Therefore, Step (3) propagates $\widetilde{O}(b)$-bit messages over the edge-disjoint fragments of size $\Theta(\sqrt{f \cdot b \cdot n})$. This can be done in $\widetilde{\Theta}(b + \sqrt{f \cdot b \cdot n})$ rounds via standard pipeline.

It remains to bound the edge congestion. By Lemma 21(3) the edge congestion of Step (0) is $\widetilde{O}(\sqrt{f \cdot n})$. Step (1) sends $\widetilde{O}(\sqrt{f \cdot n})$ messages over each edge, by Cor. 16. The edge congestion of Step (2) is bounded by $\widetilde{O}(\sqrt{f \cdot b \cdot n})$, by Lemma 11. Finally, Step (3) sends $\widetilde{O}(b)$ bits on each edge by Equation (5). Thus, overall the edge congestion is bounded by $\widetilde{O}(b + \sqrt{f \cdot b \cdot n})$.

Bonus property: Anonymous broadcast. We note that Alg. SecureEavesBroadcast can also hide from the eavesdropper, not only the identity of the broadcast message m^*, but also the identity of the sender s. The only involvement of the source s is in Step (1), i.e., in the application of the SecureMulticast algorithm of Cor. 16. Since the latter hides the identity of the source s, the final broadcast algorithm leaks no information on s, as well.

Extension to broadcast with multiple sources, proof of Cor. 7. Consider a collection of sources $S \subseteq V$, where each $s \in S$ holds a b-bit message m^*_s. For simplicity of explanation, in the following, we do not try to hide the identity[6] of S. Hence, within $O(D + |S|)$ rounds, all vertices can define an ordering on these sources, given by $s_1, \ldots, s_k$ (e.g., based on IDs).

The algorithm is based on a reduction to the (f-secure) single-source broadcast of Theorem 6 using message size of $b' = b \cdot |S|$. In the first phase, the algorithm picks some arbitrary vertex s (possibly not in S) and let it locally define a b'-bit message r^* chosen uniformly at random in $\{0,1\}^{b'}$. The source s applies Alg. SecureEavesBroadcast with the message $m^* = r^*$. This can done with $\widetilde{O}(D + \sqrt{f \cdot b \cdot |S| \cdot n})$ rounds. At this point, all the vertices share a random string r^* of b' bits, which can be locally partitioned into $|S|$ random keys $r_1, \ldots, r_{|S|} \in \{0,1\}^b$. Each r_i is used as a random key for encrypting the broadcast message of the ith source $s_i \in S$, as follows.

Each source $s_i \in S$ encrypts its b-bit broadcast message m^*_s by letting $r^*_s = m^*_s \oplus r_i$ (one-time padding). Then, we apply the *standard* (i.e., non-secure) broadcast procedure w.r.t each s_i and the message r^*_s. Using the random-delay approach of Theorem 9, all these procedures can be done in parallel within $\widetilde{O}(D + b|S|)$ rounds. As each vertex v knows r^*, it can recover the ith message by letting $m^*_s = r^*_s \oplus r_i$. The security of the algorithm simply follows by the security guarantees of the single-source broadcast algorithm of Theorem 6.

4 Forbidden-Set Broadcast

In this section, we turn to consider secure broadcast algorithms in the presence of a semi-honest adversary that controls a subset of the *vertices*. This adversary can be trusted to run the protocol honestly but aim to extract information on the vertices' input and output. Recall that in the forbidden-set broadcast problem, given is an n-graph $G = (V, E)$ with a vertex partition $R \cup F = V$, into trusted *receivers* R and untrusted vertices F, controlled by a semi-honest adversary. It is assumed that each vertex knows whether it is in R or F (hence, each vertex in R knows its neighbors in R), and that $G[R]$ is connected (which is essential). It is then required for a source vertex $s \in R$ to send a broadcast message m^* (say of $O(\log n)$-bits) to all vertices in R, while leaking no information to the eavesdropper controlling the vertices in F. That is, the collection of all messages received by the vertices in F are required to convey no information on m^*, in the information-theoretic sense.

We start by observing that the security guarantees of the unicast and multicast algorithms of Section 2 also hold in the presence of semi-honest adversaries, provided that $G[R]$ is connected. This allows us to securely exchange messages between s, t pairs in R using $O(D)$ rounds. In the full version, we show:

▷ Claim 22 (Secure Unicast and Multicast against Semi-Honest Adversaries). The security guarantees of Alg. SecureUnicast and Alg. SecureMulticast hold in the presence of semi-honest adversaries, provided that $G[R]$ is connected.

A trivial solution for the problem works by sending the message m^* over the graph $G[R]$. This, however, might possibly lead to a round complexity of $|R|$, which might be linear in n. Instead, our $\widetilde{O}(D + \sqrt{|R|})$-round algorithm is based on *using* the *untrusted* vertices for the purpose of faster communication (i.e., as relay vertices), but in a way that guarantees that these vertices still learn nothing on the secret m^*. The algorithm has three phases. First, it samples a collection of $\widetilde{O}(\sqrt{|R|})$ vertices from R, denoted as *landmarks*. The source s

[6] This can be done by small parametrization of Alg. SecureEavesBroadcast.

sends the message m^* to these landmarks using the secure multicast procedure of Claim 22 and Cor. 16. The algorithm then computes a MST T in $G[R]$, and decomposes it into a collection of tree *fragments* $\mathcal{T}$, each of size $\Theta(\sqrt{|R|})$ using Lemma 11. Note that, w.h.p., each fragment contains at least one sampled landmark. Finally, each landmark propagates m^* over the edges of its fragment, this is done in all the fragments of $\mathcal{T}$, in parallel.

Algorithm 1 ForbiddenSetBroadcast.

Input: Graph $G = (V, E)$, s holds a message m^*, a vertex partition $V = F \cup R$.
Output: All vertices in R learn the message m^*, while leaking no information to F.

- **Step (1): Multicast m^* to $\widetilde{O}(\sqrt{|R|})$ Landmarks.**
  - Sample a landmark set $L = R[p]$ for $p = \Theta(\log n/\sqrt{|R|})$.
  - Apply Alg. SecureMulticast(s, L, m^*) of Cor. 16.
- **Step (2): Tree Fragmentation**
  - Compute a spanning tree T in $G[R]$ (e.g., using the MST algorithm of [11]).
  - Apply Alg. DecomposeTree$(T, \sqrt{|R|})$ to decompose T into edge-disjoint trees, $\mathcal{T} = \{T_1, \ldots, T_\ell\}$ such that $|T_i| = \Theta(\sqrt{|R|})$ and $\bigcup_i V(T_i) = R$.
- **Step (3): Broadcast over the Fragments.**
  - For each fragment T_i, each landmark $\ell \in T_i$ broadcasts m^* over T_i.

Correctness and security. Since the size of each tree fragment T_i is $\Theta(\sqrt{|R|})$, by the Chernoff bound, we have that w.h.p. $V(T_i) \cap L \neq \emptyset$ for every $T_i \in \mathcal{T}$. As the collection of T_i trees covers all vertices in R, we get that, w.h.p., all vertices in R receive the message m^*. We now consider security and show that a semi-honest adversary controlling all vertices in $F = V \setminus R$ learns nothing on m^*. The security of Step (1) follows by Claim 22. Since the semi-honest adversary knows the graph topology, Step (2) reveals no information on the secret message m^*. Additionally, Step (3) sends messages only on edges in $G[R]$, and therefore it is secure as well.

Running time. By Chernoff, w.h.p., it holds that $|L| = O(\sqrt{|R|} \cdot \log n)$. Therefore by Cor. 16, Step (1) is implemented in $\widetilde{O}(D + \sqrt{|R|})$ rounds. The computation of T can be done in $O(D + \sqrt{|R|})$ rounds, using a standard MST procedure, e.g., applying the low-congestion shortcut framework of [11]. The fragmentation of T into the trees $\mathcal{T}$ in Step (2) takes $\widetilde{O}(\sqrt{|R|})$ rounds, using Lemma 11. Finally, the communication in each tree $T_i \in \mathcal{T}$ of Step (3) takes $O(|T_i|) = O(\sqrt{|R|})$ rounds. As all trees are edge-disjoint, this can be done in parallel over all vertices using $O(\sqrt{|R|})$ rounds. Overall, the round complexity is $\widetilde{O}(D + |\sqrt{|R|})$. This completes the proof of Theorem 8.

References

1 Rudolf Ahlswede, Ning Cai, Shuo-Yen Robert Li, and Raymond W. Yeung. Network information flow. *IEEE Trans. Inf. Theory*, 46(4):1204–1216, 2000.

2 Noga Alon, Mohsen Ghaffari, Bernhard Haeupler, and Majid Khabbazian. Broadcast throughput in radio networks: Routing vs. network coding. In Chandra Chekuri, editor, *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 1831–1843. SIAM, 2014.

3 Marshall Ball, Elette Boyle, Ran Cohen, Lisa Kohl, Tal Malkin, Pierre Meyer, and Tal Moran. Topology-hiding communication from minimal assumptions. *IACR Cryptol. ePrint Arch.*, page 388, 2021.

4 Ning Cai and Raymond W Yeung. Secure network coding. In *Proceedings IEEE International Symposium on Information Theory,*, page 323. IEEE, 2002.

5 Keren Censor-Hillel, Mohsen Ghaffari, and Fabian Kuhn. Distributed connectivity decomposition. In Magnús M. Halldórsson and Shlomi Dolev, editors, *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 156–165. ACM, 2014.

6 Keren Censor-Hillel, Bernhard Haeupler, D. Ellis Hershkowitz, and Goran Zuzic. Broadcasting in noisy radio networks. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 33–42. ACM, 2017. doi:10.1145/3087801.3087808.

7 David L Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

8 Danny Dolev, Cynthia Dwork, Orli Waarts, and Moti Yung. Perfectly secure message transmission. In *31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume I*, pages 36–45. IEEE Computer Society, 1990.

9 Jon Feldman, Tal Malkin, Cliff Stein, and Rocco A Servedio. On the capacity of secure network coding. In *Proc. 42nd Annual Allerton Conference on Communication, Control, and Computing*, pages 63–68. Cambridge University Press, 2004.

10 Mohsen Ghaffari. Near-optimal scheduling of distributed algorithms. In Chryssis Georgiou and Paul G. Spirakis, editors, *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 3–12. ACM, 2015.

11 Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks II: low-congestion shortcuts, mst, and min-cut. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 202–219. SIAM, 2016.

12 Niv Gilboa and Yuval Ishai. Compressing cryptographic resources. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 591–608. Springer, 1999.

13 Oded Goldreich. *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press, 2004. doi:10.1017/CBO9780511721656.

14 Bernhard Haeupler, David Wajc, and Goran Zuzic. Network coding gaps for completion times of multiple unicasts. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 494–505. IEEE, 2020.

15 Yael Hitron and Merav Parter. General CONGEST compilers against adversarial edges. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)*, volume 209 of *LIPIcs*, pages 24:1–24:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

16 Kamal Jain. Security based on network topology against the wiretapping attack. *IEEE Wirel. Commun.*, 11(1):68–71, 2004.

17 Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014.

18 Frank Thomson Leighton, Bruce M. Maggs, and Satish Rao. Packet routing and job-shop scheduling in O(congestion + dilation) steps. *Comb.*, 14(2):167–186, 1994.

19 Frank Thomson Leighton, Bruce M Maggs, and Satish B Rao. Packet routing and job-shop scheduling ino (congestion+ dilation) steps. *Combinatorica*, 14(2):167–186, 1994.

20 Benoît Libert, Kenneth G. Paterson, and Elizabeth A. Quaglia. Anonymous broadcast encryption: Adaptive security and efficient constructions in the standard model. In Marc Fischlin, Johannes Buchmann, and Mark Manulis, editors, *Public Key Cryptography - PKC 2012 - 15th International Conference on Practice and Theory in Public Key Cryptography, Darmstadt, Germany, May 21-23, 2012. Proceedings*, volume 7293 of *Lecture Notes in Computer Science*, pages 206–224. Springer, 2012.

21 Mahnush Movahedi, Jared Saia, and Mahdi Zamani. Secure anonymous broadcast. In Fabian Kuhn, editor, *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, volume 8784 of *Lecture Notes in Computer Science*, pages 567–568. Springer, 2014.

22 C. St.J. A. Nash-Williams. Edge-Disjoint Spanning Trees of Finite Graphs. *Journal of the London Mathematical Society*, s1-36(1):445–450, 1961.

23 Merav Parter and Eylon Yogev. Distributed algorithms made secure: A graph theoretic approach. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1693–1710. SIAM, 2019.

24 Merav Parter and Eylon Yogev. Low congestion cycle covers and their applications. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1673–1692, 2019.

25 Merav Parter and Eylon Yogev. Optimal short cycle decomposition in almost linear time. In *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, pages 89:1–89:14, 2019.

26 Merav Parter and Eylon Yogev. Optimal short cycle decomposition in almost linear time. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 89:1–89:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

27 Merav Parter and Eylon Yogev. Secure distributed computing made (nearly) optimal. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 107–116, 2019.

28 David Peleg. Time-optimal leader election in general networks. *J. Parallel Distributed Comput.*, 8(1):96–99, 1990.

29 David Peleg. *Distributed Computing: A Locality-sensitive Approach.* SIAM, 2000.

30 Christian Scheideler. *Universal routing strategies for interconnection networks*, volume 1390. Springer Science & Business Media, 1998.

31 Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

32 Aravind Srinivasan and Chung-Piaw Teo. A constant-factor approximation algorithm for packet routing and balancing local vs. global criteria. *SIAM J. Comput.*, 30(6):2051–2068, 2000.

33 Chih-Chun Wang and Minghua Chen. Sending perishable information: Coding improves delay-constrained throughput even for single unicast. *IEEE Transactions on Information Theory*, 63(1):252–279, 2016.

34 Xunrui Yin, Zongpeng Li, Yaduo Liu, and Xin Wang. A reduction approach to the multiple-unicast conjecture in network coding. *IEEE Transactions on Information Theory*, 64(6):4530–4539, 2017.

A Missing Proofs of Section 2

Proof of Claim 17. We show by induction on k that given an assignment to the random coins $\mathcal{C}_{\leq k}$ and the messages $M_{F^*}(k)$, the eavesdropper can deduce $\mathcal{T}_k(v_i)$ for every $i \geq k+1$. As the eavesdropper knows the probability distribution of $\mathcal{C}_{\leq k}$, it follows that the eavesdropper can calculate the distribution $D(k, v_i)$ for every $i \geq k+1$.

For the base case of $k = 1$, in the first round, the only vertex that can send messages is $s = v_0$. Hence, for every $i \geq 2$, the vertex v_i receives no messages in the first round and $\mathcal{T}_1(v_i) = \emptyset$. Since the eavesdropper knows the graph topology, it can deduce that $\mathcal{T}_1(v_i) = \emptyset$ (regardless of the coins $\mathcal{C}_{\leq 1}$ and the messages $M_{F^*}(1)$). Assume that given an assignment to $\mathcal{C}_{\leq k-1}$ and $M_{F^*}(k)$, the eavesdropper can deduce $\mathcal{T}_k(v_i)$, for every $i \geq k$.

We next consider round k, and a fixed assignment to $M_{F^*}(k)$ and $\mathcal{C}_{\leq k}$. Let v_i be a vertex such that $i \geq k + 1$. Since $i \geq k$, by the induction assumption the eavesdropper can deduce the tuples in $\mathcal{T}_{k-1}(v_i)$ based on $M_{F^*}(k-1) \subseteq M_{F^*}(k)$ and $\mathcal{C}_{\leq k-1} \subseteq \mathcal{C}_{\leq k}$. We are left to show the eavesdropper can learn the messages received by v_i from its neighbors in round k.

Recall that v_i has three neighbors, v_{i-1}, v_{i+1} and u. As the eavesdropper controls the edge $(u, v_i) \in F^*$, the message sent from u to v_i in round k is determined by $M_{F^*}(k)$. For $v_j \in \{v_{i-1}, v_{i+1}\}$, we note that the message v_j sent to v_i in round k is fully determined by the information received by v_j up to round $k - 1$ (i.e., $\mathcal{T}_{k-1}(v_j)$) and the random coins $\mathcal{C}_{\leq k}(v_j)$. Since $i - 1 \geq k - 1$, by the induction assumption the eavesdropper can deduce $\mathcal{T}_{k-1}(v_j)$ based on $M_{F^*}(k-1)$ and $\mathcal{C}_{\leq k-1}$. It follows that given $\mathcal{C}_{\leq k}$ and $M_{F^*}(k)$, the eavesdropper can compute the messages v_{i-1} and v_{i+1} sent to v_i in round k. The claim follows. ◀

Routing Schemes and Distance Oracles in the Hybrid Model

Fabian Kuhn
Universität Freiburg, Germany

Philipp Schneider
Universitä Freiburg, Germany

Abstract

The HYBRID model was introduced as a means for theoretical study of *distributed* networks that use various communication modes. Conceptually, it is a synchronous message passing model with a *local communication mode*, where in each round each node can send large messages to all its neighbors in a local network (a graph), and a *global communication mode*, where each node is allotted limited (polylogarithmic) bandwidth per round to communicate with *any* node in the network.

Prior work has often focused on shortest paths problems in the local network, as their global nature makes these an interesting case study how combining communication modes in the HYBRID model can overcome the individual lower bounds of either mode. In this work we consider a similar problem, namely computation of *distance oracles* and *routing schemes*. In the former, all nodes have to compute *local tables*, which allows them to look up the distance (estimates) to any target node in the local network when provided with the *label* of the target. In the latter, it suffices that nodes give the next node on an (approximately) shortest path to the target.

Our goal is to compute these local tables as fast as possible with labels as small as possible. We show that this can be done *exactly* in $\widetilde{O}(n^{1/3})$ communication rounds and labels of size $\Theta(n^{2/3})$ bits. For constant stretch approximations we achieve labels of size $O(\log n)$ in the same time. Further, as our main technical contribution, we provide computational lower bounds for a variety of problem parameters. For instance, we show that computing solutions with stretch below a certain constant takes $\widetilde{\Omega}(n^{1/3})$ rounds even for labels of size $O(n^{2/3})$.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Mathematics of computing → Graph algorithms

Keywords and phrases Distributed Computing, Graph Algorithms, Complexity Analysis

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.28

Related Version *Full Version*: https://doi.org/10.48550/arXiv.2202.06624

1 Introduction

Real networks often employ multiple communication modes. For instance, mobile devices combine high-bandwidth, short-range wireless communication with relatively low-bandwidth cellular communication (c.f., 5G [4]). Other examples are software defined networking [22] or hybrid data centers, which combine wireless and wired communication [11] or optical circuit switching and electrical packet switching [23].

In this article we utilize the theoretical abstraction of such hybrid communication networks provided by [5] which became known as *hybrid model* and was designed to reflect a high-bandwidth local communication mode and a low-bandwidth global communication mode, capturing one of the main aspects of real hybrid networks. Fundamentally, the hybrid model builds on the concept of *synchronous message passing*, a classic model to investigate round complexity in distributed systems.

© Fabian Kuhn and Philipp Schneider;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 28; pp. 28:1–28:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

▶ **Definition 1** (Synchronous Message Passing, c.f., [17]). *Let V be a set of n nodes with unique identifiers $ID : V \to [n] \overset{\text{Def}}{=} \{1, \ldots, n\}$. Time is slotted into discrete rounds consisting of the following steps. First, all nodes receive the set of messages addressed to them in the last round. Second, nodes conduct computations based on their current state and the set of received messages to compute their new state (randomized algorithms also include the result of some random function). Third, based on the new state the next set of messages is sent.*

Synchronous message passing has a clear focus on investigating round complexity, i.e., the number of communication rounds required to solve a problem with an input distributed over all nodes. For this purpose, nodes are usually assumed to be computationally unbounded. Occasionally this model is "overexploited", e.g., nodes are supposed solve $\mathcal{NP}$-complete problems on their local data, however we will refrain from that. The hybrid model places additional restrictions on the size of message and which nodes can exchange them.

▶ **Definition 2** (Hybrid model [5]). *The* $\mathsf{HYBRID}(\lambda, \gamma)$ *model is a synchronous message passing model (Def. 1), subject to the following restrictions.* Local mode: *nodes may send a message per round of maximum size λ bits to each of their neighbors in a connected graph.* Global mode: *nodes can send and receive messages of total size at most γ bits per round to/from any other node(s) in the network. If the restrictions are not adhered to, then a strong adversary selects the messages that are delivered.*

The parameter spectrum of the $\mathsf{HYBRID}(\lambda, \gamma)$ model covers the standard models LOCAL, $\mathsf{CONGEST}$, CLIQUE (aka "Congested Clique") and NCC ("Node Capacitated Clique", [5]) as marginal cases: LOCAL: $\lambda = \infty, \gamma = 0$, $\mathsf{CONGEST}$: $\lambda = O(\log n), \gamma = 0$, CLIQUE (+ Lenzen's routing protocol [16]): $\lambda = 0, \gamma = n \log n$, NCC: $\lambda = 0, \gamma = O(\log^2 n)$. Given the ramifications of investigating $\mathsf{HYBRID}(\lambda, \gamma)$ in its entirety, we narrow our scope (for our upper bounds) to a particular parametrization that pushes both communication modes to one extreme end of the spectrum. Following the arguments of [5] we leave the size of local messages unrestricted (modeling high local bandwidth) and allow only polylog n bits of global communication per node per round (modeling severely restricted global bandwidth). Formally, we define the "standard" hybrid model as combination of the standard LOCAL and NCC models: $\mathsf{HYBRID} := \mathsf{HYBRID}\big(\infty, O(\log^2 n)\big)$. Our lower bounds are given for the more general $\mathsf{HYBRID}(\infty, \gamma)$ model, which implies lower bounds for the weaker HYBRID model.

A fundamental aspect of the Internet Protocol is packet forwarding, where every node has to compute a routing function, which – when combined with target-specific information stored in the packet header – must indicate the neighbor which the packet has to be forwarded to such that it reaches its intended destination. A correct *routing scheme* consists of these routing functions and a unique *label* per node, such that the packet forwarding procedure induces a path in the network from *any* source node to *any* destination node specified by the corresponding label attached to the packet.

We distinguish *stateful* and *stateless* routing schemes. In the former, routing can be based on additional information accumulated in the packet header as the packet is forwarded, whereas in the latter routing decisions are completely oblivious to the previous routing path. A related problem is the computation of distance oracles, which is similar to the all pairs shortest paths (APSP) problem. Each node must compute an oracle function that provides the distance (or an estimate) to any other node when provided with the corresponding label. Formal definitions are given in Section 1.3 (Def. 3, 4 and 5).

The first goal is to gather the required data for labels, routing and oracle functions in as few rounds as possible. This is particularly important for dynamic or unreliable networks where changes in distances or topology necessitates (frequent) re-computation. In this

work we allow to relabel nodes (a.k.a. *labeling scheme*), where the new labels may contain information that helps with distance estimation and routing decisions. This gives rise to our second goal; keeping node labels small.[1] The third goal is to speed up the actual packet forwarding process to minimize latency and alleviate congestion. Given a graph with edge weights corresponding to (e.g.) link-latencies, we want to minimize the largest detour any packet takes relative to the corresponding shortest path. This is also known as *stretch*. Analogously, for distance oracles we want to minimize the worst estimation error relative to the true distance.

In this work we are interested in solving the above problems in a distributed setting (c.f., Definition 1). This has particular importance given the distributed nature of many real networks where routing problems are relevant (most prominently, the Internet) and where providing a centralized view of the whole network is prohibitively expensive. Given that usually vast quantities of data must be routed, we interested in computing routing schemes and distance oracles for the local communication network, which offers the majority of the throughput, whereas the throughput of global network is considered negligible. The global mode is used to speed up the computation of routing schemes (significantly, as we will see in this work) and can also be used to send the (relatively small) labels to a source, which can then be stored at that node for the duration of a session.

From an algorithmic standpoint, computing routing schemes and distance oracles is an inherently *global* problem. That is, allowing only local communication (i.e., the LOCAL model) it takes $\Omega(n)$ rounds to accomplish this.[2] A similar observation can be made for the global communication mode (NCC model). If we are only allowed to use global communication and each node initially only knows its incident edges in the local network, it takes $\widetilde{\Omega}(n)$ [3] rounds to compute routing schemes and distance oracles. This article addresses the question whether the combination of the two communication modes in the HYBRID model can overcome the $\widetilde{\Omega}(n)$ lower bound of the individual modes LOCAL and NCC.

Our answer to this is two-pronged. First we show that indeed, we can compute routing schemes and distance oracles significantly faster, for instance, $\widetilde{O}(n^{1/3})$ rounds and labels of size $\Theta(n^{2/3})$ suffice for an exact solution (c.f., Theorem 27). This separates these problems from the APSP problem, which can be used to solve our problems at hand, but has a $\widetilde{\Omega}(\sqrt{n})$ lower bound even for extremely crude approximations [5]. Second, we show that the HYBRID model is not arbitrarily powerful by giving polynomial lower bounds for these problems (depending on the stretch) that hold even for relatively large labels and unbounded local memory. For instance, we show that it takes $\widetilde{\Omega}(n^{1/3})$ rounds to solve either problem exactly, even for unweighted graphs and labels of size $O(n^{2/3})$ (c.f., Theorem 14). We provide nuanced results, depending on stretch and the type of problem, summarized in the following.

1.1 Contributions and Overview

Table 1 gives a simplified overview of our complexity results for the various forms of routing scheme and distance oracle problems. Our main contributions revolve around computational lower bounds for computing distance oracles and stateless and stateful routing schemes in

---

[1] In other settings, the amount of information stored at nodes for routing and distance estimation is considered, as well. Since the nodes in our model are computationally unbounded we do not focus on that. Our lower bounds have also no restriction on the local information.

[2] Note that any graph problem can be solved in $O(n)$ rounds in LOCAL by collecting the graph and solving the problem locally at some node. This makes global problems uninteresting for the LOCAL model, unless communication restrictions are increased (c.f., CONGEST) or decreased (c.f., HYBRID).

[3] The $\widetilde{O}(\cdot)$ notation suppresses multiplicative terms that are polylogarithmic in n.

the HYBRID model. Lower bounds for approximations are summarized in the first three groups of Table 1. We also consider lower bounds on unweighted graphs, see the first row of the fourth group of Table 1. These lower bounds hold regardless of the allowed local memory and hold for randomized algorithms with constant success probability.

Furthermore, we give upper bounds for the HYBRID model, summarized in the last two groups of Table 1. The implied computational upper bounds for the exact and approximate problem variants are almost tight with some of our corresponding lower bounds (more on that further below). In the following paragraphs we aim to give some intuitive understanding into how our techniques work by shedding most of the proof-details and occasionally pointing out how the results are generalized in the main part. We also show how the techniques used in this work relate to – and are distinct from – prior work.

In the main part, the results on lower bounds are formulated for the more general $\mathsf{HYBRID}(\infty,\gamma)$ model (which therefore hold for $\mathsf{HYBRID}(\lambda,\gamma)$), and thus γ appears as parameter. For instance, the precise bound on unweighted graphs or for approximations on weighted graphs is $\Omega(n^{1/3}/\gamma^{1/3})$ rounds for labels of size up to $c{\cdot}n^{2/3}{\cdot}\gamma^{1/3}$ for some $c > 0$ (c.f., Theorem 14). That is, we get a polynomial lower bound for the $\mathsf{HYBRID}(\infty,\gamma)$ model for any $\gamma = n^t$ for constant $t < 1$. For easier readability we plug in the "standard" HYBRID model with $\gamma = \widetilde{O}(1)$, which lets us hide γ using the $\widetilde{\Omega}$ notation.

Table 1 Selected and simplified contributions of this paper.

problem	stretch	complexity	label-size	reference
distance oracles	$3-\varepsilon$	$\widetilde{\Omega}(n^{1/3})$	$O(n^{2/3})^{\dagger}$	Theorem 20
	ℓ	$\widetilde{\Omega}\big(n^{1/f(\ell)}\big)^{\ddagger}$	$O\big(n^{2/f(\ell)}\big)^{\dagger\ddagger}$	Theorem 20, 21
stateless routing schemes	$\sqrt{3}-\varepsilon$	$\widetilde{\Omega}(n^{1/3})$	$O(n^{2/3})^{\dagger}$	Theorem 23
	$\sqrt{5}-\varepsilon$	$\widetilde{\Omega}(n^{1/5})$	$O(n^{2/5})^{\dagger}$	Theorem 23
	$1+\sqrt{2}-\varepsilon$	$\widetilde{\Omega}(n^{1/7})$	$O(n^{2/7})^{\dagger}$	Theorem 23
stateful routing schemes	$\sqrt{2}-\varepsilon$	$\widetilde{\Omega}(n^{1/3})$	$O(n^{2/3})^{\dagger}$	Theorem 25
	$\frac{5}{3}-\varepsilon$	$\widetilde{\Omega}(n^{1/5})$	$O(n^{2/5})^{\dagger}$	Theorem 25
	$\frac{7}{4}-\varepsilon$	$\widetilde{\Omega}(n^{1/7})$	$O(n^{2/7})^{\dagger}$	Theorem 25
	≈ 1.78	$\widetilde{\Omega}(n^{1/11})$	$O(n^{2/11})$	Theorem 25
all of the above on *unweighted graphs*	exact	$\widetilde{\Omega}(n^{1/3})$	$O(n^{2/3})^{\dagger}$	Theorem 14
	$1+\varepsilon$	$\widetilde{O}(n^{1/3}/\varepsilon)$	$\Theta(\log n)$	Theorem 28
all of the above on *weighted graphs*	exact	$\widetilde{O}(n^{1/3})$	$\Theta(n^{2/3})$	Theorem 27
	3	$\widetilde{O}(n^{1/3})$	$\Theta(\log n)$	Theorem 28

†) The lower bound on round complexity holds for node labels of at most that size.
‡) For some function $f(\ell)$ that is linear in ℓ.

Communication Lower Bound in HYBRID. The proofs of lower bounds are based on an intermediate problem which describes the complexity of communicating information between distinct node sets in the HYBRID model, which we can then translate into the realm of classic information theory. We sketch the idea bottom up, neglecting generalizations and most of the details. We start out with a two party communication problem, where Alice is given the state of some random variable X and needs to communicate it to Bob (c.f., Definition 8). Any communication protocol that achieves this needs to communicate $H(X)$ (Shannon entropy of X, see Def. 29) bits in expectation (c.f., Corollary 31), which is a consequence of the source coding theorem (replicated in Lemma 30).

In Section 2 we reduce this to the HYBRID setting into what we call the *node communication problem*. There, we have two sets of nodes A and B, where nodes in A "collectively know" the state of some random variable X and need to communicate it to B (for more precise information see Definition 7). We show a reduction where a HYBRID algorithm that solves the node communication problem on sets A and B, which are at sufficiently large distance in the local graph, can be used to derive a protocol for the two party communication problem (c.f., Lemma 9). We conclude that for sets A, B with distance at least h it takes $\widetilde{\Omega}\big(\min(H(X)/n, h)\big)$ rounds to solve this problem (Theorem 10).

The node communication problem extends on [13, 5] and can be seen as a natural intermediate problem that lies at the core of many interesting problems in the HYBRID(∞, γ) model and might be useful as black-box for other reductions. A major difference to [13] is that there, two party set disjointness is reduced to a distributed decision problem, whereas we show that learning super-linear information from distant parts of a graph is hard in the HYBRID model. The APSP lower bound of [5] uses the observation that a large number of distances has to be learned by a single node. However, labels (even as small as $O(\log n)$) prohibit this idea in our setting, as these must be treated as "free information" that can provide these distances. On a technical level we generalize the approach for the HYBRID(∞, γ) model and strengthen the proof for randomized protocols for *any* constant success probability (see Appendix A).

Lower Bounds on Unweighted Graphs. In Section 3 we construct a reduction from the node communication problem to distance oracle and routing scheme computation in the HYBRID model. The goal is to encode some random variable X with large entropy (super-linear in n) into some randomized part of our local communication graph such that some node set A knows X by vicinity. We construct such a graph Γ (see Figure 1a) from the complete bipartite graph $G_{k,k} = (A, E)$ and a (i.i.d.) random k^2-bit-string $X = (x_e)_{e \in E}$ with $H(X) = \Theta(k^2)$. Then an edge $e \in E$ of $G_{k,k}$ is present in Γ iff $x_e = 1$.

The nodes in A collectively know X since they are incident to the edges sampled from $G_{k,k}$. We designate k nodes of A (one side of the bipartition in $G_{k,k}$) as the "target nodes". We connect each target with a path of length h to one of k "source nodes" which takes the role of B (see Figure 1a). We show that if the nodes in B learn the distances to the target nodes they also learn about the (non-)existence of the edges sampled from $G_{k,k}$ and can conclude the state of X and thus have solved the node communication problem. Balancing the parameters k and h we conclude that $\widetilde{\Omega}(n^{1/3})$ rounds of communication must have taken place to solve the distance oracle problem exactly (Theorem 14).

For routing schemes we have to adapt the graph Γ. We add a slightly longer alternative route from sources to targets (Figure 1a, left side) and show that the state of X can be concluded from the first routing decision the sources have to make, i.e., whether this alternative path is used or not. One caveat is that the nodes are only supposed to give a distance or make a correct routing decision when also provided with the target-label. We choose the labels sufficiently small such that the "free information", given in form of the labels of all targets, is negligible. Still, labels up to size $O(n^{2/3})$ do not change the above narrative, cf. Theorem 14.

Lower Bounds for Approximations. In Section 4 we show how to use graph weights to obtain lower bounds for approximations. We replace $G_{k,k}$ with a balanced, bipartite graph $G = (A, E)$ with k nodes and girth ℓ (length of the shortest cycle in G). As before, the existence of an edge $e \in E$ in Γ is determined by a random bit string $X = (x_e)_{e \in E}$,

c.f., Figure 1b. If some edge $e \in E$ is not in Γ, then the detour between endpoints of e using other edges sampled from G is at least $\ell-1$ edges, which translates into almost the same multiplicative detour, by assigning large weights to those edges. Any algorithm for distance oracles with stretch slightly smaller than $\ell-1$ can then be used to solve the node communication problem.

To maximize $H(X)$ we need to maximize the density of G. However, it is known that girth and density of a graph are opposing goals: a graph with girth $2g+1$ can have at most $O\big(n^{1+1/g}\big)$ edges (c.f., [2], simplified in Lemma 34). This inherently limits the amount of information we can encode in Γ and we show in Lemma 18 how graph density affects lower bounds for the node communication problem. The good news is, that for some girth values, graphs that achieve their theoretical density limit actually exist and have been constructed (c.f., [6, 19], simplified form given in Lemma 35). For higher girth values, graphs that come close to that limit are known (c.f., [15], simplified form in Lemma 36). Utilizing these graphs we prove *polynomial* lower bounds for the *distance oracle* problem for small stretch values (c.f., Theorem 20) but also for arbitrary constant stretch (c.f., Theorem 21). Theorem 21 is heavily parametrized, but to sum it up in a simpler way: for any constant stretch ℓ we attain a polynomial lower bound of $\widetilde{\Omega}(n^{1/f(\ell)})$, that is, $f(\ell)$ is constant as well (roughly $f(\ell) \approx \frac{3}{2}\ell$).

A major part is dedicated to lower bounds for approximate stateless and stateful *routing schemes* (c.f., Definitions 4, 5). Here, the introduction of weights, stretch and girth introduces considerable complexity when using our techniques. The main reason for this is that in routing problems a wrong routing decision at the source can often still be completed into a routing path of relatively good quality, even more so for stateful routing, where a packet may "backtrack" to try different paths (for those reasons the lower bounds are also more limited in terms of stretch). We carefully optimize our bad case graph such that it maximizes the stretch for certain round complexities. The results are given in the second and third group of Table 1 with details in Theorems 23 and 25.

Our reductions from the node communication problem to various bad graph instances for distance oracle and routing scheme problems are distinct from those in [13], which shows a $\widetilde{\Omega}(n^{1/3})$ lower bound for computing the diameter that reduces from the two party set-disjointness problem. Our bad case graphs are also distinct from those for APSP in [5], which creates a bottleneck for a single node, that must learn $\widetilde{\Omega}(n)$ bits (which does not work for labeling schemes). Lower bounds for the same problem in the CONGEST model are provided by [12]. Here, the goal is to construct a graph G that has a communication bottleneck (usually a small cut) and small diameter D_G as the $\Omega(D_G)$ lower bound is trivial. By contrast, as the HYBRID model contains LOCAL, small cuts do not help us and we have a trivial $O(D_G)$ upper bound, thus we need to look at graphs with relatively large diameter.

Upper Bounds. For our computational upper bounds (given in Section 5) we show how to reduce the computation of distance oracles and routing schemes for *general graphs* in the HYBRID model to shortest path problems. In particular we draw on fast solutions for the so called *random sources shortest paths problem* (RSSP) [7], where all nodes must learn their distance to a set of i.i.d. randomly sampled nodes, say S. After solving RSSP, our strategy is to use the distance between a node u and the nodes in S as its label $\lambda(u)$.

Roughly speaking, provided that u is sufficiently "far away", a node v can combine $\lambda(u)$ with its own distances to S to compute its distance (estimate) to u. If u is "close" then we can use the local network to compute the distance directly. While this gives us only distance oracles, routing schemes can also be derived. Simply speaking, we can always send a packet to a neighbor that has the best distance (estimate) to u, although some further care must be taken for approximations. Note that this process is oblivious to previous routing decisions so the obtained routing scheme is stateless (c.f., Definition 4).

A trade-off arises from the local exploration around nodes and the global computation depending on the size of S (since we solve RSSP on S), which balances out to a round complexity of $\widetilde{O}(n^{1/3})$ with $|S| \in \widetilde{O}(n^{2/3})$. We obtain exact algorithms (distance oracles and routing schemes) with labels of size $\Theta(n^{2/3})$ (however we can further decrease label size to $\Theta(n^{2/3-\zeta})$ at a cost of $\widetilde{O}(n^{1/3+\zeta})$ rounds, c.f., Theorem 27). Note that this is tight up to polylog n factors as is shown by the corresponding lower bound in Table 1 group 4 line 1 (which holds even on unweighted graphs).

For smaller labels we obtain a 3-approximation on weighted graphs and a $(1+\varepsilon)$ approximation on unweighted graphs in $\widetilde{O}(n^{1/3})$ rounds (for constant $\varepsilon > 0$) with labels of size $O(\log n)$ (c.f., Theorem 28), which is as small as labels can asymptotically be to be able to identify the destination. Compare this to our lower bounds: even much larger labels of size $\Theta(n^{2/3})$ do *not* help to improve the stretch by much, as this still takes $\widetilde{\Omega}(n^{1/3})$ rounds for stretch of $3-\varepsilon$ for distance oracles on weighted graphs, and stretch 1 on unweighted graphs (see Table 1).

Our upper bounds separate the distance oracle (and routing) problem from the related *all pairs shortest paths* (APSP) problem, where nodes must give their distance to all other nodes *without* using labels and where a $\widetilde{\Omega}(n^{1/2})$ lower bound is known even for stretch up to some $\alpha \in \widetilde{\Theta}(n^{1/2})$ [13]. This separation is not the case in the LOCAL and NCC models, where either problem has round complexity $\widetilde{\Theta}(n)$ rounds in general. Our results show that labels of limited size of $O(\log n)$ bits helps to significantly speed up computing (approximate) distances to all destinations in the HYBRID model.

1.2 Related Work

There was an early effort to approach hybrid networks from a theoretic angle [1], with a conceptually different model. Research on the current take of the HYBRID model was initiated by [5] in the context of shortest paths problems, which most of the research has focused on so far. As shortest paths problems problems are closely related, we give a brief account of the recent developments. An overview of distance oracles and routing schemes in other models is provided in the full version of this article [14].

Shortest Paths in the Hybrid Model. [5] introduced an information dissemination scheme to efficiently broadcast small messages to all nodes in the network. Using this protocol, they derive various solutions for shortest paths problems. For instance, for SSSP: a $(1+\varepsilon)$ stretch, $\widetilde{O}(n^{1/3})$-round algorithm and a $(1/\varepsilon)^{O(1/\varepsilon)}$-stretch, $\widetilde{O}(n^{\varepsilon})$-round algorithm. Further, an approximation of APSP with stretch 3 in $\widetilde{O}(n^{1/2})$ rounds, which closely matches their $\widetilde{\Omega}(n^{1/2})$ lower bound (which holds for much larger stretch). Subsequently, [13] introduced a protocol to efficiently uni-cast small messages between dedicated source-target pairs in the HYBRID model, which they use to solve APSP and SSSP exactly in $\widetilde{O}(n^{1/2})$ and $\widetilde{O}(n^{2/5})$ rounds, respectively. For computing the diameter they provide algorithms (e.g., a $3/2+\varepsilon$ approximation in $\widetilde{O}(n^{1/3})$ rounds) and a $\widetilde{\Omega}(n^{1/3})$ lower bound. [7] combines the techniques of [13] with a densitiy sensitive approach, to solve $n^{1/3}$-SSP (thus SSSP) exactly and compute a $(1+\varepsilon)$-approximation of the diameter in $\widetilde{O}(n^{1/3})$ rounds. [8] uses density awareness in a different way to improve SSSP to $\widetilde{O}(n^{5/17})$ rounds for a small stretch of $(1+\varepsilon)$. For classes of sparse graphs (e.g., cactus graphs) [10] demonstrates that exact solutions in $\widetilde{O}(1)$ rounds are possible even in the harsher hybrid combination CONGEST and NCC. The recent result of [24] implies an $(1+\varepsilon)$ approximation of SSSP in $\widetilde{O}(1)$ rounds in *general* graphs in the hybrid CONGEST and NCC regime, since their partwise aggregation model can be simulated efficiently in the HYBRID model as shown by [3]. The article by [3] also derandomized the dissemination protocol of [5] to obtain a deterministic APSP-algorithm with stretch $\frac{\log n}{\log \log n}$ in $\widetilde{O}(n^{1/2})$ rounds.

1.3 Preliminaries

General Definitions. The scope of this paper is solving graph problems, typically in the undirected local graph $G = (V, E)$. Edges have weights $w : E \to [W]$, where W is at most polynomial in n, thus the weight of an edge and of a simple path fits into a $O(\log n)$ bit message (whereas define $\log := \log_2$). Graph G is considered unweighted if $W = 1$. Let $w(P) = \sum_{e \in P} w(e)$ denote the length of a path $P \subseteq E$. Then the *distance* between two nodes $u, v \in V$ is $d_G(u, v) := \min_{u\text{-}v\text{-path } P} w(P)$. A path with smallest length between two nodes is called a *shortest path*. Let $|P|$ be the number of edges (or *hops*) of a path P. The *hop-distance* between two nodes u and v is defined as: $\text{hop}_G(u, v) := \min_{u\text{-}v\text{-path } P} |P|$. We generalize this to sets $U, W \subseteq V$ (whereas $\text{hop}_G(v, v) := 0$): $\text{hop}_G(U, W) := \min_{u \in U, w \in W} \text{hop}_G(u, w)$. The *diameter* of G is defined as: $D_G := \max_{u,v \in V} \text{hop}_G(u, v)$. Let the *$h$-hop distance* from u to v be: $d_{G,h}(u, v) := \min_{u\text{-}v\text{-path } P, |P| \le h} w(P)$. If there is no u-v path P with $|P| \le h$ we define $d_h(u, v) := \infty$. We drop the subscript G, if G is clear from the context. We consider the following problems:

▶ **Definition 3** (Distance Oracles). *Every node $v \in V$ of a graph $G = (V, E)$ needs to compute a label $\lambda(v)$ and an oracle function $o_v : \lambda(V) \to \mathbb{N}$, such that $o_v(\lambda(u)) \ge d(u, v)$ for all $u \in V$. An oracle function o_v is an (α, β)-approximation if $o_v(\lambda(u)) \le \alpha \cdot d(u, v) + \beta$ for all $u, v \in V$, that is, α, β are the multiplicative and additive approximation error, respectively. We speak of a stretch of α in case of an $(\alpha, 0)$-approximation. If the stretch is one, we call o_v exact.*

▶ **Definition 4** (Stateless Routing Scheme). *Every node $v \in V$ of a graph $G = (V, E)$ needs to learn a label $\lambda(v)$ and a routing function (sometimes called "table") $\rho_v : \lambda(V) \to N(v) \cup \{v\}$ where $N(v)$ are adjacent nodes of v in G (whereas we formally set $\rho_v(\lambda(v)) := v$). The functions ρ_v must fulfill the following correctness condition. Let $v_0 := v$ and recursively define $v_i := \rho_{v_{i-1}}(\lambda(u))$. Then the routing functions $\rho_v, v \in V$ must satisfy $v_h = u$ for some $h \in \mathbb{N}$. Let $P_\rho(u, v)$ be the path induced by the visited nodes $v_0, \dots, v_h$. We call ρ an (α, β)-approximation if $w(P_\rho(u, v)) \le \alpha d(u, v) + \beta$ for all $u, v \in V$.*

▶ **Definition 5** (Stateful Routing Scheme). *This is mostly defined as in the stateless case, with the difference that ρ_v can additionally depend on the information gathered along the path that has already been visited by a packet (which would be stored in its header). Note that the routing path defined by such a function ρ might have loops.*

▶ **Definition 6** (Randomized Graph Algorithms). *We say that an algorithm has success probability p, if it succeeds with probability at least p on every possible input graph (however, some of our results are restricted to unweighted input graphs). Specifically, for our upper bounds we aim for success with high probability (w.h.p.), which means with success probability at least $1 - \frac{1}{n^c}$ for arbitrary constant $c > 0$.*

2 Node Communication Problem

In this section we create an "information bottleneck" in the HYBRID model between two (distant) parts of the local communication graph. We do this for the more general HYBRID(∞, γ) model, where we have a global communication bandwidth of γ bits per node per round, which also has the advantage of avoiding logarithmic terms and O-notation as long as possible (recall that HYBRID = HYBRID$(\infty, O(\log^2 n))$).

We first introduce some definitions. We say that the nodes from some set $A \subseteq V$ *collectively know* the state of a random variable X, if its state can be derived from the information that the nodes A have. Or, in terms of information theory, given the state or

input S_A of all nodes A (interpreted as a random variable), then the conditional entropy $H(X|S_A)$ (see Definition 29), also known as the amount of new information of X provided that S_A is already known, is zero. Similarly, we say that the state of X is *unknown* to $B \subseteq V$, if the initial information of the nodes B does not induce any knowledge on the outcome of X. Or formally, that for the state S_B of the nodes B we have that $H(X|S_B) = H(X)$, meaning that all information in X is new even if S_B is known.

▶ **Definition 7** (Node Communication Problem). *Let $G = (V, E)$ be some graph. Let $A, B \subset V$ be disjoint sets of nodes and $h := hop(A, B)$. Furthermore, let X be a random variable whose state is collectively known by the nodes A but unknown to any set of nodes disjoint from A. An algorithm $\mathcal{A}$ solves the* node communication problem *if the nodes in B collectively know the state of X after $\mathcal{A}$ terminates. We say $\mathcal{A}$ has success probability p if $\mathcal{A}$ solves the problem with probability at least p for* any *state X can take (in line with our Definition 6 of success probability for graph algorithms).*

The goal of Lemma 9 is to reduce a more basic communication problem, for which we can provide lower bounds using information theory (c.f., Appendix A) to the node communication problem. Analogously to node sets, we define that Alice knows some random variable X, which is unknown to Bob as follows. Given that S_{Alice} and S_{Bob} are their respective inputs then we have $H(X|S_{\text{Alice}}) = 0$ and $H(X|S_{\text{Bob}}) = H(X)$.

▶ **Definition 8** (Two Party Communication Problem). *Given two computationally unbounded parties, Alice and Bob, where initially Alice knows the state of some random variable X which is unknown to Bob. A communication protocol $\mathcal{P}$ is said to solve that problem if after its execution Bob can derive the state of X from the transcript of all exchanged messages. Performance is measured in the length of the transcript in bits. We say $\mathcal{P}$ has success probability p if $\mathcal{P}$ solves the problem with probability at least p for* any *state X can take.*

The reduction from the 2-party communication problem to the node communication problem uses the following simulation argument: Alice and Bob can together simulate a $\mathsf{HYBRID}(\infty, \gamma)$ model algorithm for the node communication problem and use it solve the 2-party communication problem. The proof is deferred to Appendix C.

▶ **Lemma 9.** *Any algorithm $\mathcal{A}$ that solves the* node communication problem *(Def. 7) in the $\mathsf{HYBRID}(\infty, \gamma)$ model on some local graph $G = (V, E)$ with $n = |V|$ and $A, B \subset V$ in $T < h = hop(A, B)$ rounds with success probability p can be used to obtain a protocol $\mathcal{P}$ that solves the two party communication problem (Def. 8) with the same success probability p and transcript length at most $T \cdot n \cdot \gamma$.*

We plug in the lower bound for the 2-party communication problem (c.f. Lemma 32 in Appendix A) to derive a lower bound for the node communication problem. Note that this theorem only depends on the hop distance h between A, B, the entropy of X and the number of nodes n and is otherwise agnostic to the local graph. Also note that a lower bound that holds in expectation is also a worst case lower bound.

▶ **Theorem 10.** *Any algorithm that solves the node communication problem (Def. 7) on some n-node graph in the $\mathsf{HYBRID}(\infty, \gamma)$ model with success probability at least p, takes at least $\min\big(\frac{pH(X)-1}{n \cdot \gamma}, h\big)$ rounds in expectation, where $H(X)$ denotes the entropy of X.*

Proof. We have to show that a randomized, $\mathsf{HYBRID}(\infty, \gamma)$ algorithm $\mathcal{A}$ that solves the node communication problem in *less* than h rounds with success probability p takes at least $\frac{pH(X)-1}{n \cdot \gamma}$ rounds. Presume, for a contradiction, that $\mathcal{A}$ has an expected running time $T < h$ *and* $T < \frac{pH(X)-1}{n \cdot \gamma}$. This implies $T \cdot n \cdot \gamma < p \cdot H(X) - 1$.

Invoking Lemma 9 gives us a protocol $\mathcal{P}$ with the same success probability p and with a transcript of length at most $T \cdot n \cdot \gamma$. With the inequality above, this means in the protocol $\mathcal{P}$, Alice sends *less* than $p \cdot H(X) - 1$ bits to Bob in expectation. This contradicts the fact that $p \cdot H(X) - 1$ is a lower bound for this due to Appendix A Lemma 32. ◀

We have to accommodate the fact that in the routing problem or distance oracle problem, the nodes have to give a distance estimation or next routing neighbor only when provided with the label of the target node. Therefore we have to slightly amend Theorem 10, which will later allow us to argue that even if we assume that nodes have advance knowledge of a selection of sufficiently small labels, the lower bound will not change asymptotically.

▶ **Corollary 11.** *If A is allowed to communicate y bits to B for free, then any algorithm that solves the node communication problem on some n-node graph (Def. 7) in the* HYBRID(∞, γ) *model with success probability at least p, takes at least $\min\left(\frac{pH(X)-1-y}{n\cdot\gamma}, h\right)$ rounds in expectation (i.e., also in the worst case).*

3 Lower Bounds For Unweighted Graphs

In this and the following section we aim to reduce from the node communication problem in the HYBRID(∞, γ) model given in Definition 7, to the problem of computing routing tables or distance oracles, which works as follows. We define a graph $\Gamma = (V_\Gamma, E_\Gamma)$ such that, first, the solution of the routing or distance oracle problems informs a subset $B \subset V_\Gamma$ about the exact state of some random variable X that is encoded by the subgraph induced by $A \subset V_\Gamma$. Second, X has a large entropy (we aim for super-linear in n). And third, the distance $hop(A, B)$ between both sets is sufficiently large.

▶ **Definition 12.** *Let $X = (x_{ij})_{i,j\in[k]} \in \{0,1\}^{k^2}$ be a bit sequence of length k^2. Let $\Gamma = (V_\Gamma, E_\Gamma)$ (shown in Figure 1a) be an unweighted graph with source nodes $s_1, \dots, s_k \in V_\Gamma$, transit nodes $u_1, \dots, u_k \in V_\Gamma$ and target nodes $t_1, \dots, t_k \in V_\Gamma$. Each source s_i has a path of length h hops to the transit nodes u_i. We have an edge between u_i and t_j if and only if $x_{ij} = 1$. Additionally, there are two nodes $v, v' \in V_\Gamma$ connected by a path of h hops. The nodes v and v' have an edge to each source s_i or target t_i, respectively (Figure 1a).*

This construction has the following properties.

(1) The distance from source s_i to t_j is larger for $x_{ij} = 0$ than for $x_{ij} = 1$ (as shown by the subsequent Lemma 13).

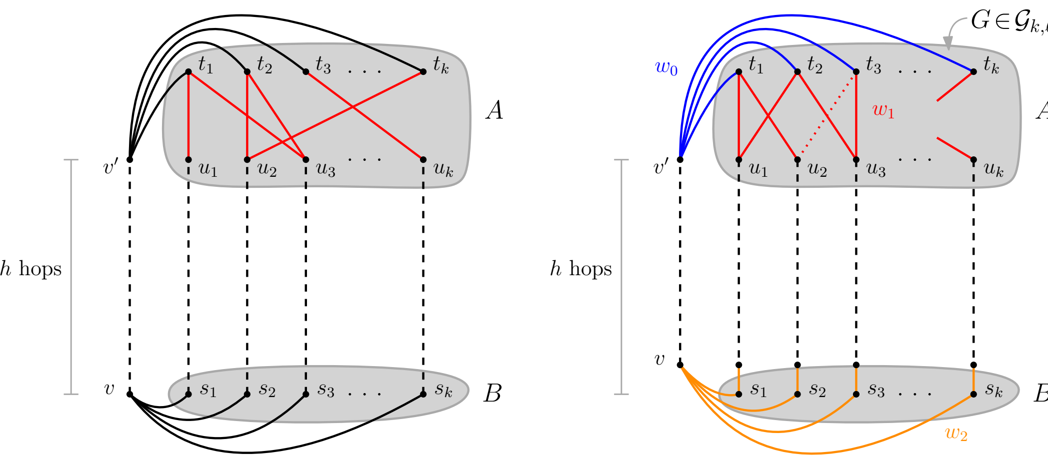
(2) For all $i, j \in [k]$, independently set $x_{ij} = 1$ with probability $\frac{1}{2}$, else $x_{ij} = 0$. This maximizes $H(X) = -k^2 \cdot \frac{\log(1/2)}{2} = \frac{k^2}{2}$.

(3) Let $A = \{u_1, \dots, u_k, t_1, \dots, t_k\}$, $B = \{s_1, \dots, s_k\}$, i.e., $hop(A, B) = h$.

▶ **Lemma 13.** *If $x_{ij} = 1$ then $d(s_i, t_j) = h+1$ and the shortest s_i-t_j-path contains v, else $d(s_i, t_j) = h+2$ and it does not contain v.*

Proof. Any path from s_i to t_j has to cross the vertex cut $U := \{u_1, \dots, u_k, v'\}$ (c.f., Figure 1a). Such a path has to include a path of length h to reach a node of U, as well as an additional edge connecting U to t_j and therefore $d(s_i, t_j) \geq h + 1$. However, we also have $d(s_i, t_j) \leq h + 2$, due to the path along the nodes $s_i, v, \dots, v', t_j$ (c.f., Figure 1a) that has length $h + 2$.

If $x_{ij} = 1$, i.e., $\{u_i, t_j\} \in E$, then the path along the nodes $s_i, \dots, u_i, t_j$ has length $h + 1$. Note that all nodes in $U \setminus \{u_i, v'\}$ are at distance at least $h + 2$ from s_i (c.f., Figure 1a), so every path via one of the nodes $U \setminus \{u_i, v'\}$ has distance at least $h + 3$. In the case $x_{ij} = 0$, i.e., $\{u_i, t_j\} \notin E_\Gamma$, this is also true for the path via u_i and the only path with distance $h + 2$ is the one via v'. ◀



(a) Graph $\Gamma = (V_\Gamma, E_\Gamma)$. Bit string $X = (x_{ij}) \in \{0,1\}^{k^2}$ determines red edges. E.g., $\{u_1, t_2\} \notin E_\Gamma$ and $\{u_1, t_3\} \in E_\Gamma$ means $x_{12} = 0$ and $x_{13} = 1$.

(b) Graph Γ constructed around $G = (A, E) \in \mathcal{G}_{k,\ell}$ with $m := |E|$ edges (red) and from $X = (x_e)_{e \in E} \in \{0,1\}^m$, where $e \in E$ is part of Γ iff $x_e = 1$.

Figure 1 Lower bound graphs. Unweighted (left) and weighted (right).

The idea to prove the next theorem is that if the nodes in B learn the distance to the nodes in $t_1, \dots, t_k$, then their combined knowledge can be used to infer the state of the random string X that is collectively known by the nodes in A. The proof the applies Theorem 10 and we maximize the lower bound by balancing k (which givers the size of $H(X)$) and h. Formally, we apply Corollary 11 to account for the labels $\lambda(t_1), \dots, \lambda(t_k)$ that are given "for free" to the nodes in B, which gives us an upper bound on their size for which the lower bound in round complexity still holds. The full proof is given in [14].

▶ **Theorem 14.** *Even on* unweighted graphs*, any randomized algorithm that computes exact (stateless or stateful) routing schemes or distance oracles in the* HYBRID(∞, γ) *model with constant success probability takes* $\Omega(n^{1/3}/\gamma^{1/3})$ *rounds. This holds for labels of size up to* $c \cdot n^{2/3} \cdot \gamma^{1/3}$ *(for a fixed constant $c > 0$).*

4 Lower Bounds for Approximations

Our next construction relies on the existence of families of graphs that have high girth and maintain relatively high density. We modify the basic construction above, essentially by replacing the upper part of Γ with a random selection of edges from a graph of that family (and also making Γ weighted). Besides high density we require the following.

▶ **Definition 15.** *$\mathcal{G}_{k,\ell}$ is a graph family, s.t. for all $G = (A, E) \in \mathcal{G}_{k,\ell}$: (i) $|A| = 2k$, (ii) G has (even) girth at least ℓ, (iii) G is balanced and bipartite.*

Removing an edge from $G \in \mathcal{G}_{k,\ell}$ incurs a large detour of at least $\ell - 1$ hops between the endpoints of that edge, since otherwise there would be a cycle shorter than ℓ in G. This observation is often used to prove certain bounds for low stretch subgraphs (one prominent example is the lower bound on the size of low stretch spanners). and can be exploited to introduce a stretch into our lower bound construction. We construct this formally as follows (however, first consulting Figure 1b will presumably be more helpful to the reader).

▶ **Definition 16.** *Let $G = (A, E) \in \mathcal{G}_{k,\ell}$ with $m := |E|$ edges and let $\{u_1, \dots, u_k\} \cup \{t_1, \dots, t_k\} = A$ be the bipartition of G. Graph $\Gamma = (V_\Gamma, E_\Gamma)$ (shown in Figure 1b) has a similar structure as the unweighted construction (Def. 12), where the main difference is the way how the nodes $\{u_1, \dots, u_k\} \cup \{t_1, \dots, t_k\}$ are connected by edges in Γ.*

Let $X = (x_e)_{e \in E} \in \{0,1\}^m$ be a bit string of length $m = |E|$, i.e., each bit x_e corresponds to an edge e of G. For each u_i, t_j we have $\{u_i, t_j\} \in E_\Gamma$, if and only if $\{u_i, t_j\} \in E$ and $x_{\{u_i,t_j\}} = 1$. In a slight change from the previous construction, we make the path from v to v' of hop length $h-1$. The weights of Γ are assigned as follows. Edges between the node v' and some t_j have weight w_0. Edges between nodes u_i, t_j have weight w_1. Edges incident to some s_i have weight w_2.

We have the following properties.

(1) Let $e = \{u_i, t_j\} \in E$. w_0, w_1, w_1 can be chosen s.t. $d(s_i, t_j)$ is much longer for $x_e = 0$ than for $x_e = 1$ (c.f., Lemma 17).

(2) For each edge $e \in E$ of G, set $x_e = 1$ i.i.d. with probability $\frac{1}{2}$, else $x_e = 0$. This maximizes the entropy $H(X) = \frac{m}{2}$.

(3) For nodes A of G and $B := \{s_1, \ldots, s_k\}$ we have $\text{hop}(A, B) = h$.

We observe that the distances $d(s_i, t_j)$ between nodes s_i, t_j with $e = \{u_i, t_j\} \in E$ depend on whether e is in Γ ($x_e = 1$), or not ($x_e = 0$), the proof is deferred to Appendix D. Conceptually, we later choose weights $w_1 \ll w_0$, so that x_e induces a large difference in $d(s_i, t_j)$.

▶ **Lemma 17.** *Consider Γ (Def. 16), constructed from $G = (A, E) \in \mathcal{G}_{k,\ell}$ and X. Let $w_1 < w_0 < (\ell - 1)w_1$. Let $e = \{u_i, t_j\} \in E$. Then we have:*

(i) *The shortest s_i-t_j-path contains v if and only if $x_e = 0$.*

(ii) *If $x_e = 1$ then $d(s_i, t_j) = w_2 + w_1 + h - 1$.*
If $x_e = 0$, then $d(s_i, t_j) = w_2 + w_0 + h - 1$.

For the reduction from the node communication problem to our concrete routing and distance oracle problems, we start with a technical lemma that analyzes the running time of any algorithm $\mathcal{A}$ that solves the node communication problem in Γ for the dedicated node sets A, B and the random variable X from which Γ is constructed.

In particular, we express the lower bound from Theorem 10 as function of $n := |V_\Gamma|$, the density of $G \in \mathcal{G}_{k,\ell}$ given by a parameter δ and the global communication capacity γ. The lemma is the result of balancing a trade off between the distance $h = \text{hop}(A, B)$ and the number of nodes $\Theta(k)$ of G (which governs the entropy $H(X) = \Theta(k^{1+\delta})$ when the density of G is fixed).For the proof details see [14].

▶ **Lemma 18.** *Consider Γ constructed from random variable X and $G = (A, E) \in \mathcal{G}_{k,\ell}$ (Def. 16) with $|E| = \Theta(k^{1+\delta})$ edges (for $\delta > 0$ and k of our choosing). Let $\mathcal{A}$ be an algorithm that solves the node communication problem on Γ with X, node sets $A, B \subset V_\Gamma$ and $h = hop(A, B)$ in the $\mathsf{HYBRID}(\infty, \gamma)$ model (all parameters as in Def. 16). We can choose $k = \Theta\big(\frac{n}{h}\big)$ such that $\mathcal{A}$ takes $\Omega\big((\frac{n^\delta}{\gamma})^{\frac{1}{2+\delta}}\big)$ rounds. There exists a constant $c > 0$ such that this holds even when we allow exchanging $c \cdot k^{1+\delta}$ bits from A to B for free.*

4.1 Distance Oracles

The first lower bound with stretch is for the distance oracle problem. The idea is as follows. In case there is a direct edge $e = \{u_i, t_j\}$ (i.e., $x_e = 1$), the distance from s_i to t_j is almost $\ell-1$ times shorter, than if that is not the case. Hence, by learning an approximation of $d(s_i, t_j)$ with a stretch slightly lower than $\ell-1$, the node s_i can conclude if e exists or not, i.e., if $x_e = 1$ or $x_e = 0$. Hence the nodes $B = \{s_1, \ldots, s_k\}$ collectively learn the random variable X and thus solve the node communication problem.

This lemma is kept general such that we can plug in any graph with density parameter δ and girth ℓ. Note that the girth ℓ fundamentally limits the density parameter δ; the correspondence between the two is roughly $\delta \in O\big(\frac{1}{\ell}\big)$ as shown in Appendix B. For a more intuitive understanding we suggest plugging in the complete bipartite graph $G_{k,k}$ which has girth $\ell = 4$ and $\Theta(k^2)$ edges (i.e., density parameter $\delta = 1$). The proof appears in [14].

▶ **Lemma 19.** *Consider Γ constructed from $G = (A, E) \in \mathcal{G}_{k,\ell}$ with $|E| = \Theta\big(k^{1+\delta}\big)$ edges for some $\delta > 0$. Any algorithm that solves the* distance oracle problem *on Γ with stretch $\alpha_\ell = \ell-1-\varepsilon$ (for any const. $\varepsilon > 0$) and constant success probability in the* HYBRID(∞, γ) *model takes $\Omega\big((\frac{n^\delta}{\gamma})^{\frac{1}{2+\delta}}\big)$ rounds, for labels up to size $c \cdot n^{\frac{2\delta}{2+\delta}} \cdot \gamma^{\frac{\delta}{2+\delta}}$ (for a fixed const. $c > 0$).*

It remains to insert graphs $G \in \mathcal{G}_{k,\ell}$ into Lemma 19. We aim for graphs $G = (V, E) \in \mathcal{G}_{k,\ell}$ with $|E| \in \Theta\big(k^{1+\delta}\big)$ that maximize both girth ℓ and density parameter δ. As outlined in Appendix B, these are opposing objectives, and for even girth $\ell \geq 4$ we know that $\delta \in O\big(\frac{2}{\ell-2}\big)$ (from applying Lemma 34 on uneven girth $\ell - 1$). Bipartite graphs of girth ℓ that reach $\delta \in \Theta\big(\frac{2}{\ell-2}\big)$ can be constructed for small girth ℓ (summarized in Lemma 38) from which we obtain Theorem 20. But for higher girth we have to settle for δ below this threshold (see Lemma 39), this is reflected in Theorem 21.

▶ **Theorem 20.** *Any algorithm that solves the* distance oracle problem *in the* HYBRID(∞, γ) *model with constant success probability with*

- *stretch $3-\varepsilon$ takes $\Omega\big((\frac{n}{\gamma})^{\frac{1}{3}}\big)$ rounds for label size $\leq c \cdot (n^2\gamma)^{\frac{1}{3}}$*
- *stretch $5-\varepsilon$ takes $\Omega\big(\frac{n^{1/5}}{\gamma^{2/5}}\big)$ rounds for label size $\leq c \cdot (n^2\gamma)^{\frac{1}{5}}$*
- *stretch $7-\varepsilon$ takes $\Omega\big(\frac{n^{1/7}}{\gamma^{3/7}}\big)$ rounds for label size $\leq c \cdot (n^2\gamma)^{\frac{1}{7}}$*
- *stretch $11-\varepsilon$ takes $\Omega\big(\frac{n^{1/11}}{\gamma^{5/11}}\big)$ rounds for label size $\leq c \cdot (n^2\gamma)^{\frac{1}{11}}$*

for any const. $\varepsilon > 0$ and a fixed const. $c > 0$.

Proof. By Lemma 38 there are bipartite, balanced graphs with girth $\ell \in \{4, 6, 8, 12\}$ and $\Theta(n^{1+\frac{2}{\ell-2}})$ edges, thus $\delta(\ell) = \frac{2}{\ell-2}$. In particular, we have $\delta(4) = 1$, $\delta(6) = \frac{1}{2}$, $\delta(8) = \frac{1}{3}$, $\delta(12) = \frac{1}{5}$, which yield the desired results when plugged into Lemma 19. ◀

Applying Lemma 19 on the densest known graphs with larger girth (see Lemma 39), we obtain the subsequent theorem. The parametrization is complex due to a case distinction in Lemma 39, the upshot is that for any constant stretch and sufficiently small γ we still get polynomial (in n) lower bounds for labels up to some polynomial size (in n).

▶ **Theorem 21.** *Any algorithm that solves the* distance oracle problem *in the* HYBRID(∞, γ) *model with constant success probability for any const. $\varepsilon > 0$ and a fixed const. $c > 0$, with*

- *stretch $\ell-1-\varepsilon$ for $\ell \geq 14$ with $\ell \equiv 2 \bmod 4$ takes $\Omega\Big(n^{\frac{2}{3\ell-8}} / \gamma^{\frac{3\ell-10}{6\ell-6}}\Big)$ rounds for label size $\leq c \cdot n^{4/(3\ell-8)} \cdot \gamma^{2/(3\ell-8)}$*
- *stretch $\ell-1-\varepsilon$ for $\ell \geq 16$ with $\ell \equiv 0 \bmod 4$ takes $\Omega\Big(n^{\frac{2}{3\ell-10}} / \gamma^{\frac{3\ell-12}{6\ell-8}}\Big)$ rounds for label size $\leq c \cdot n^{4/(3\ell-10)} \cdot \gamma^{2/(3\ell-10)}$.*

Proof. By Lemma 39 there are bipartite, balanced graphs with even girth $\ell \geq 14$ that have (i) $\Theta(n^{1+\frac{4}{3\ell-10}})$ edges if $\ell \equiv 2 \bmod 4$, or (ii) $\Theta(n^{1+\frac{4}{3\ell-12}})$ edges if $\ell \equiv 0 \bmod 4$. Thus in case (i) we have $\delta(\ell) = \frac{4}{3\ell-10}$ and in case (ii) $\delta(\ell) = \frac{4}{3\ell-12}$. Plugging $\delta(\ell)$ into Lemma 19 gives the desired result. ◀

4.2 Stateless Routing Scheme

For lower bounds of routing schemes we exploit the observation that for an edge $e = \{s_i, t_j\} \in E$ the node s_i learns about the existence of e in Γ, i.e., whether $x_e = 0$ or $x_e = 1$, from the decision to send a packet with destination t_j first to v or not. More precisely, we show that $x_e = 0$ if and only if v is the first routing neighbor for the packet with destination t_j.

However, we have to decrease the stretch of our lower bound in order that this works. The main obstacle is that the decision of s_i to send a packet with target t_j directly towards u_i instead of node v (left path) does not impact the distance of the routing path that one can still obtain by that much.

In particular, in the case of *stateless* routing, a packet that travels from s_i to u_i and finds that the direct edge $\{u_i, t_j\}$ does *not* exist, could still use any other edge $\{u_i, t_p\}$ and then the two edges $\{t_p, v'\}, \{v', t_j\}$ to get to t_j (e.g., in Figure 1b from s_2 to t_3). This would mislead s_i as the first routing node was *not* v, yet $x_e = 0$.

The target is to prohibit this and some other troublesome routing options by making them exceed the stretch guarantee. However, this gives us additional restrictions that dominate the resulting system of inequalities for higher girth ℓ of G, in particular we gain no improvement in the stretch for $\ell \geq 8$. The proof is deferred to [14].

▶ **Lemma 22.** *Consider Γ constructed from $G = (A, E) \in \mathcal{G}_{k,\ell}$ with $|E| = \Theta(k^{1+\delta})$ edges for some $\delta > 0$. For any constant $\varepsilon > 0$ let $\alpha_\ell = \sqrt{\ell-1}-\varepsilon$ for $\ell \leq 6$ and $\alpha_\ell = 1+\sqrt{2}-\varepsilon$ for $\ell \geq 8$. Any algorithm that computes a* stateless routing scheme *on Γ with stretch α_ℓ and constant success probability in the* HYBRID(∞, γ) *model takes $\Omega\big((\frac{n^\delta}{\gamma})^{\frac{1}{2+\delta}}\big)$ rounds. This holds for labels of size $c \cdot n^{\frac{2\delta}{2+\delta}} \gamma^{\frac{\delta}{2+\delta}}$ and fixed constant $c > 0$.*

We plug graphs $G \in \mathcal{G}_{k,\ell}$ into Lemma 22. Since in this case we get no improvements in the stretch for girth $\ell \geq 8$ it suffices to apply Lemma 38. Beside the changed values for the stretch, the proof is the same as that of Theorem 20, we just have to use the corresponding values of δ from Lemma 38 for $\ell = 4, 6, 8$.

▶ **Theorem 23.** *Any algorithm that solves the* stateless routing problem *in the* HYBRID(∞, γ) *model with constant success probability with*

- *stretch $\sqrt{3}-\varepsilon$ takes $\Omega\big((\frac{n}{\gamma})^{\frac{1}{3}}\big)$ rounds for label size $\leq c \cdot (n^2\gamma)^{\frac{1}{3}}$*
- *stretch $\sqrt{5}-\varepsilon$ takes $\Omega\big(\frac{n^{1/5}}{\gamma^{2/5}}\big)$ rounds for label size $\leq c \cdot (n^2\gamma)^{\frac{1}{5}}$*
- *stretch $1+\sqrt{2}-\varepsilon$ takes $\Omega\big(\frac{n^{1/7}}{\gamma^{3/7}}\big)$ rounds for label size $\leq c \cdot (n^2\gamma)^{\frac{1}{7}}$*

for any const. $\varepsilon > 0$ and a fixed const. $c > 0$.

4.3 Stateful Routing Scheme

We obtain similar lower bound results for the approximate *stateful* routing problem, however with even smaller stretch. Recall that in the stateful version the problem is relaxed in the sense that a routing decision may also depend on the information a packet has gathered along the previous routing path.

Since this permits loops in the routing path, it opens up additional options for routing a packet from s_i to t_j that we need to prohibit. For instance, a packet could first travel to u_i, then check if the direct edge $\{u_i, t_j\}$ is present, and if not travel back to s_i to take the shorter route via v instead. Note that this path has the same number of red and blue edges as the shortest path directly to v and then to t_j (c.f. Figure 1b).

The trick is to make the weight w_2 (orange edges) of all incident edges of s_i more expensive, such that revisiting s_i breaks the approximation guarantee. This again forces the source s_i to make the correct decision with the first node it routes the packet to, which renders the ability to travel in loops and learn along the way useless. We carefully optimize the involved parameters to obtain the following lemma (the details are given in [14]).

▶ **Lemma 24.** *Consider Γ constructed from $G = (A, E) \in \mathcal{G}_{k,\ell}$ with $|E| = \Theta(k^{1+\delta})$ edges for some $\delta > 0$. For any constant $\varepsilon > 0$ let $\alpha_4 = \sqrt{2} - \varepsilon$, $\alpha_6 = \frac{5}{3} - \varepsilon$, $\alpha_8 = \frac{7}{4} - \varepsilon$. For $\ell \geq 10$ let $\alpha_\ell = \frac{3+\sqrt{17}}{4} - \varepsilon \approx 1.78$. Any algorithm that computes a* stateful routing scheme *on Γ with stretch α_ℓ and constant success probability in the* HYBRID(∞, γ) *model takes $\Omega\big((\frac{n^\delta}{\gamma})^{\frac{1}{2+\delta}}\big)$ rounds. This holds for labels of size $c \cdot n^{\frac{2\delta}{2+\delta}} \cdot \gamma^{\frac{\delta}{2+\delta}}$ and fixed constant $c > 0$.*

Again, our actual lower bounds come from inserting graphs $G \in \mathcal{G}_{k,\ell}$ into Lemma 22. Our best stretch is obtained for $\ell = 10$, but unfortunately we have a gap for that value in Lemma 38. Therefore, for the largest stretch value we use a graph $G \in \mathcal{G}_{k,12} \subseteq \mathcal{G}_{k,10}$, which has the drawback of not being as dense. Aside from different stretch values, the proof follows that of Theorem 20, by inserting the values of δ from Lemma 38 for $\ell = 4, 6, 8, 12$.

▶ **Theorem 25.** *Any algorithm that solves the* stateful routing problem *in the* HYBRID(∞, γ) *model with constant success probability with*

- *stretch* $\sqrt{2} - \varepsilon$ *takes* $\Omega\big((\frac{n}{\gamma})^{\frac{1}{3}}\big)$ *rounds for label size* $\leq c \cdot (n^2\gamma)^{\frac{1}{3}}$
- *stretch* $\frac{5}{3} - \varepsilon$ *takes* $\Omega\big(\frac{n^{1/5}}{\gamma^{2/5}}\big)$ *rounds for label size* $\leq c \cdot (n^2\gamma)^{\frac{1}{5}}$
- *stretch* $\frac{7}{4} - \varepsilon$ *takes* $\Omega\big(\frac{n^{1/7}}{\gamma^{3/7}}\big)$ *rounds for label size* $\leq c \cdot (n^2\gamma)^{\frac{1}{7}}$
- *stretch* $\frac{3+\sqrt{17}}{4} - \varepsilon$ *takes* $\Omega\big(\frac{n^{1/11}}{\gamma^{5/11}}\big)$ *rounds for label size* $\leq c \cdot (n^2\gamma)^{\frac{1}{11}}$

for any const. $\varepsilon > 0$ *and a fixed const.* $c > 0$.

5 Upper Bounds

In this section, we sketch our algorithms for computing routing schemes and distance oracles in the HYBRID model. Due to space limitations, the details are given in the full version [14], where we will also show that distance oracles imply *stateless* routing schemes with the same label-size, stretch and asymptotic round complexity, thus we concentrate on the former. We combine two techniques to compute exact distance oracles, namely skeleton graphs and fast algorithms for the random sources shortest paths (RSSP) problem, where all nodes need to determine their distance to a set of nodes that was sampled i.i.d. from V.

Skeleton graphs were first used by [20] and became one of the main tools used in the context of shortest path algorithms in the HYBRID model (c.f., [5, 13, 7]). Simply speaking, a skeleton consists of a (usually relatively small) set of nodes sampled with some probability $\frac{1}{x}$ and virtual edges formed between sampled nodes at most $h \in \widetilde{O}(x)$ hops apart. The main property is that for any $u, v \in V$, there is a sampled node at least every h hops on a shortest path from u to v w.h.p. Here, we only require this sampling property, the explicit construction of the skeleton graph occurs under the hood when solving the RSSP problem. The following result is known.

▶ **Lemma 26** (c.f., [7]). *There is a* HYBRID *algorithm that solves the RSSP problem for sampling probability* $1/x$ *for* $x \geq 1$ *exactly and w.h.p. in* $\widetilde{O}(n^{1/3} + n/x^2)$ *rounds.*

Our algorithm to compute distance oracles roughly consists of the following steps. First, we sample the nodes for the skeleton graph S i.i.d. with probability $\frac{1}{x}$. Then we do a local exploration to depth $h \in \widetilde{O}(x)$. If our target node has a shortest path with less or equal h hops, distance decisions can be made based on the so acquired local knowledge.

For shortest paths from v to u with more than h hops we do the following. We first solve the RSSP problem using Lemma 26. Then each node v puts the resulting distance to all skeleton nodes in S within h hops into its label . Since there is a skeleton node on a shortest path from v to u w.h.p., v can reconstruct the distance to u given the label $\lambda(u)$ as follows.

$$o_v(\lambda(u)) = \min\Big(d_h(v, u), \min_{s \in S} d(v, s) + d(s, u)\Big)$$

It remains to balance the trade-off in round complexity of $h \in \widetilde{O}(x)$ between the local search of h rounds and the HYBRID computation of RSSP distances (see Lemma 26), which is optimal for $x \in \widetilde{O}(n^{1/3})$. Note that the size of labels scales in the number of sampled nodes, thus $O\big(\frac{n}{x}\big) = O(n^{2/3})$, however, by deviating from the optimal round complexity, we can decrease the label size. The following proof of the following theorem is given in [14].

▶ **Theorem 27.** *For any* $\zeta \geq 0$ exact *distance oracles and* stateless *routing schemes with labels of size* $O(n^{\frac{2}{3}-\zeta})$ *bits can be computed in* $\widetilde{O}(n^{\frac{1}{3}+\zeta})$ *rounds in the* HYBRID *model w.h.p.*

We can also trade higher stretch for smaller labels. Given that each node u only puts its distance to the *closest* skeleton node into its label $\lambda(u)$ (which must be within h hops due to the sampling property) we can still recover distance oracles with stretch 3 on weighted and $(1+\varepsilon)$ on unweighted graphs with label size only $O(\log n)$. For details of the exact procedure and the proof of the following theorem see [14].

▶ **Theorem 28.** *Distance oracles and* stateless *routing schemes with label-size* $O(\log n)$ *can be computed in* HYBRID *w.h.p. and*
- *stretch 3 in* $\widetilde{O}(n^{1/3})$ *rounds on weighted graphs,*
- *stretch* $1+\varepsilon$ *for* $0<\varepsilon\leq 1$ *in* $\widetilde{O}\big(\frac{n^{1/3}}{\varepsilon}\big)$ *rounds on unweighted graphs.*

References

1 Yehuda Afek, Gad M. Landau, Baruch Schieber, and Moti Yung. The power of multimedia: Combining point-to-point and multiaccess networks. *Information and Computation*, 84(1):97–118, January 1990. doi:10.1016/0890-5401(90)90035-G.

2 Noga Alon, Shlomo Hoory, and Nathan Linial. The moore bound for irregular graphs. *Graphs and Combinatorics*, 18, September 2001. doi:10.1007/s003730200002.

3 Ioannis Anagnostides and Themis Gouleakis. Deterministic Distributed Algorithms and Lower Bounds in the Hybrid Model. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DISC.2021.5.

4 Arash Asadi, Vincenzo Mancuso, and Rohit Gupta. An sdr-based experimental study of outband d2d communications. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, 2016. doi:10.1109/INFOCOM.2016.7524372.

5 John Augustine, Kristian Hinnenthal, Fabian Kuhn, Christian Scheideler, and Philipp Schneider. Shortest paths in a hybrid network model. In *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '20, pages 1280–1299, USA, 2020. Society for Industrial and Applied Mathematics.

6 Clark T. Benson. Minimal regular graphs of girths eight and twelve. *Canadian Journal of Mathematics*, 18:1091–1094, 1966. doi:10.4153/CJM-1966-109-8.

7 Keren Censor-Hillel, Dean Leitersdorf, and Volodymyr Polosukhin. Distance Computations in the Hybrid Network Model via Oracle Simulations. In Markus Bläser and Benjamin Monmege, editors, *38th International Symposium on Theoretical Aspects of Computer Science (STACS 2021)*, volume 187 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.STACS.2021.21.

8 Keren Censor-Hillel, Dean Leitersdorf, and Volodymyr Polosukhin. On sparsity awareness in distributed computations. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21, pages 151–161, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3409964.3461798.

9 Paul Erdős and Miklós Simonovits. Compactness results in extremal graph theory. *Comb*, 2(3):275–288, 1982.

10 Michael Feldmann, Kristian Hinnenthal, and Christian Scheideler. Fast hybrid network algorithms for shortest paths in sparse graphs. In *Proc. of the 24th International Conference on Principles of Distributed Systems (OPODIS 2020)*, pages 31:1–31:16, 2020. doi:10.4230/LIPIcs.OPODIS.2020.31.

11 Kai Han, Zhiming Hu, Jun Luo, and Liu Xiang. Rush: Routing and scheduling for hybrid data center networks. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 415–423, 2015. doi:10.1109/INFOCOM.2015.7218407.

12 Taisuke Izumi and Roger Wattenhofer. Time lower bounds for distributed distance oracles. In Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro, editors, *Principles of Distributed Systems*, pages 60–75, Cham, 2014. Springer International Publishing.

13 Fabian Kuhn and Philipp Schneider. Computing shortest paths and diameter in the hybrid network model. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, pages 109–118, New York, NY, USA, July 2020. Association for Computing Machinery. doi:10.1145/3382734.3405719.

14 Fabian Kuhn and Philipp Schneider. Routing schemes and distance oracles in the hybrid model. *arXiv preprint*, 2022. arXiv:2202.06624.

15 Felix Lazebnik, Vasiliy A. Ustimenko, and Andrew J. Woldar. Upper bounds on the order of cages. *the electronic journal of combinatorics*, pages R13–R13, 1997.

16 Christoph Lenzen. Optimal deterministic routing and sorting on the congested clique. In *Proc. 32nd Symp. on Principles of Distr. Comp. (PODC)*, pages 42–50, 2013.

17 Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.

18 Claude Elwood Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

19 Robert Singleton. On minimal graphs of maximum even girth. *J. Comb. Theory*, 1:306–332, 1966.

20 Jeffrey D. Ullman and Mihalis Yannakakis. High-probability parallel transitive-closure algorithms. *Journal on Computing*, 20(1):100–125, 1991.

21 Jacques Verstraëte. Extremal problems for cycles in graphs. In *Recent trends in combinatorics*, pages 83–116. Springer, 2016.

22 Stefano Vissicchio, Laurent Vanbever, and Olivier Bonaventure. Opportunities and research challenges of hybrid software defined networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):70–75, April 2014. doi:10.1145/2602204.2602216.

23 Guohui Wang, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, T.S. Eugene Ng, Michael Kozuch, and Michael Ryan. C-through: Part-time optics in data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 327–338, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1851182.1851222.

24 Goran Zuzic, Gramoz Goranci, Mingquan Ye, Bernhard Haeupler, and Xiaorui Sun. Universally-optimal distributed shortest paths and transshipment via graph-based $\ell 1$-oblivious routing. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2549–2579. SIAM, 2022.

A Information Theoretic Concepts

The Shannon entropy of a random variable X can be thought of as the average information conveyed by a realization of X and is defined as follows.

▶ **Definition 29** (Entropy, c.f., [18]). *The Shannon entropy of a random variable* $X:\Omega\to S$ *is defined as* $H(X) := -\sum_{x\in S}\mathbb{P}(X=x)\log\big(\mathbb{P}(X=x)\big)$. *For two random variables* X,Y *the joint entropy* $H(X,Y)$ *is defined as the entropy of* (X,Y). *The conditional entropy is* $H(X|Y) = H(X,Y) - H(Y)$. *The transinformation is defined as* $I(X;Y) = H(X) - H(Y|X)$.

The Entropy $H(X)$ gives a lower bound for expected number of bits required for encoding the state of a random variable. This is entailed by Shannon's [18] source coding theorem.

▶ **Lemma 30** (c.f., [18]). *Given a random variable* X *with outcomes from some set* S *and an uniquely decodable code* $f : S \to \{0,1\}^*$ *with expected code length* $\mathbb{E}(|f(X)|)$. *Then* $\mathbb{E}(|f(X)|) \geq H(X)$.

In particular, in a two party communication setting (see Definition 8) this implies that $H(X)$ constitutes a lower bound for the worst case number of bits that have to be transmitted from one party that knows the state of X to some party that needs to learn it.

▶ **Corollary 31.** *Bob must receive at least $H(X)$ bits from Alice in expectation, as part of any protocol solving the two party communication problem (Def. 8).*

Proof. Assume, for a contradiction, that we have a protocol $\mathcal{P}$ in which sending *less* than $H(X)$ bits from Alice to Bob always suffices to solve the two party communication problem. Clearly, for any two possible outcomes $x_1, x_2 \in S$ of X, the transcript of the communication occurring between Alice and Bob must be different as otherwise Bob would not be able to distinguish x_1 from x_2. But then we could use the transcript of $\mathcal{P}$ for any given outcome of $x \in S$ of X as uniquely decodable code for x of expected length less than $H(X)$, a contradiction to Lemma 30. ◀

Using information theoretic concepts, the above statement generalizes for a protocol that has a probability of at least p that Bob can successfully decode the state of X after it is terminates.

▶ **Lemma 32.** *Bob must receive at least $p \cdot H(X) - 1$ bits from Alice in expectation, as part of any protocol that solves the two party communication problem (c.f., Def. 8) with probability at least p.*

Proof. We assume that the random variable X has a finite number of outcomes (which is sufficient for our purposes), i.e., $X \in \{x_1, \dots, x_k\}$ for some $k \in \mathbb{N}$. Assuming the outcome $X = x_i$, let y_i be the output that Bob makes after the randomized communication protocol terminates. Then

$$y_i = \begin{cases} x_i, & \text{with probability } p_i \\ x_j \text{ and } j \neq i, & \text{with probability } (1 - p_i), \end{cases}$$

where $p \leq p_i \leq 1$. That means we have another random variable Y dependent on X, which describes Bob's guess about the state of X. It remains to prove that the information about X that is still contained in Y, is large. This is known as the *transinformation* $I(X;Y)$ and since Bob "learns" the state of Y, at least $I(X;Y)$ must have been transmitted from Alice to Bob. In particular, we want to show $I(X;Y) \geq p \cdot H(X) - 1$. The transinformation (see Def. 29) can also be written as follows

$$I(X;Y) = \sum_{i,j \in [k]} \mathbb{P}(X = x_i, Y = x_j) \cdot \log \frac{\mathbb{P}(X = x_i, Y = x_j)}{\mathbb{P}(X = x_i)\mathbb{P}(Y = x_j)}$$

Analyzing this directly is tricky since the output distribution of Y for the second case, where $Y \neq X$, is not specified (and can not be made such, without losing the generality of the claim). So we have to take a detour by defining a third random variable Z that tells us if the protocol was successful.

$$Z = \begin{cases} 1, & \text{if } y_i = x_i \\ 0, & \text{else.} \end{cases}$$

To simplify the analysis of the transinformation we assume that Bob gets to know Z "for free" and since $H(Z) \leq 1$ the additional information about X from learning Y is not significantly reduced. Formally, we first show $I(X;Y) \geq I(X;Y,Z) - H(Z)$ which allows us to analyze $I(X;Y,Z)$ instead.

The conditional entropy $H(A|B)$ describes the amount of "new" information in some random variable A given that we already know random variable B. In the following steps we will use the fact that $H(Z|X,Y) = 0$ since Z is functionally dependent on X and Y and we will the chain rule of entropy $H(A,B) = H(A|B)+H(B)$. We plug this into the alternative characterization of transinformation

$$\begin{aligned}
I(X;Y,Z) &= H(X) - H(X|Y,Z) && \textit{def. of } I(X;Y,Z)\\
&= H(X) - H(X,Z|Y) + H(Z|Y) && \textit{chain rule}\\
&= H(X) - H(Z|X,Y) - H(X|Y) + H(Z|Y) && \textit{chain rule}\\
&= H(X) - H(X|Y) + H(Z|Y) && H(Z|X,Y) = 0\\
&= I(X;Y) + H(Z|Y) && \textit{def. of } I(X;Y)\\
&\leq I(X;Y) + H(Z).
\end{aligned}$$

This implies $I(X;Y) \geq I(X;Y,Z) - H(Z)$, and it remains to show that $I(X;Y,Z)$ is large. The random variable Z helps in the following way. For any x_i we have

$$\mathbb{P}(Y{=}x_i, Z{=}1) = p_i \cdot \mathbb{P}(X{=}x_i) = \mathbb{P}(X{=}x_i, Y{=}x_i, Z{=}1),$$

since $Z = 1$ means that $Y = x_i$ is only possible if $X = x_i$. We obtain

$$\begin{aligned}
I(X;Y,Z) &= \sum_{i,j\in[k],z\in\{0,1\}} \mathbb{P}(X{=}x_i, Y{=}x_j, Z{=}z) \cdot \log \frac{\mathbb{P}(X{=}x_i, Y{=}x_j, Z{=}z)}{\mathbb{P}(X{=}x_i) \cdot \mathbb{P}(Y{=}x_j, Z{=}z)}\\
&\geq \sum_{i\in[k]} \mathbb{P}(X{=}x_i, Y{=}x_i, Z{=}1) \cdot \log \frac{\mathbb{P}(X{=}x_i, Y{=}x_i, Z{=}1)}{\mathbb{P}(X{=}x_i) \cdot \mathbb{P}(Y{=}x_i, Z{=}1)}\\
&= \sum_{i\in[k]} p_i \cdot \mathbb{P}(X{=}x_i) \cdot \log \frac{1}{\mathbb{P}(X{=}x_i)}\\
&\geq\ p \cdot \sum_{i\in[k]} \mathbb{P}(X{=}x_i) \cdot \log \frac{1}{\mathbb{P}(X{=}x_i)} = p \cdot H(X)
\end{aligned}$$

Finally, we have $I(X;Y) \geq I(X;Y,Z) - H(Z) \geq p \cdot H(X) - 1$. ◀

B Density of Bounded Girth Graphs

We reproduce a few known and conjectured results from extremal graph theory, in particular that the number of edges in cycle-free graphs can be bounded from above and below. We are going to formulate these results in the context and granularity that we require in this article (neglecting constants, in particular). First, there is a long standing conjecture from Erdős and Simonovits [9].[4]

▶ **Conjecture 33** (by [9]). *For any $k \in \mathbb{N}$, there is an n-node graph with girth $\geq 2k{+}1$ and $\Theta(n^{1+\frac{1}{k}})$ edges.*

It is known that a graph with average degree d and girth $2k + 1$ has $n \in \Omega(d^k)$ nodes due to [2]. This translates into the following lemma:

[4] [9] states in Conjecture 5 that there are graphs without cycles of a fixed length with the claimed density, and conjectures that the same holds for excluding smaller cycles as well (below Theorem 2 of [9]).

▶ **Lemma 34** (c.f., [2]). *Any n-node graph with girth at least $2k+1, k \in \mathbb{N}$ has at most $O(n^{1+\frac{1}{k}})$ edges.*

Conjecture 33 is known to be true for some parameters of k due to [19] and [6].

▶ **Lemma 35** (c.f., [19, 6]). *For $k = 2,3,5$ there are n-node graphs with girth $2k+1$ and $\Theta(n^{1+\frac{1}{k}})$ edges.*

There are more general lower bounds for graphs for arbitrary girth by [15] which the survey [21] summarizes as follows:

▶ **Lemma 36** (c.f., [15, 21]). *For any $k \geq 2$ there is a n-node graph with girth $2k+1$ and $\Theta(n^{1+\frac{2}{3k-2}})$ edges if k is even, and $\Theta(n^{1+\frac{2}{3k-3}})$ if k is odd.*

Above we mention only uneven girth, whereas in this paper we are mostly interested in (balanced) bipartite graphs which naturally have even girth. Note that given a graph with girth $2k+1$, one easily obtains a balanced, bipartite graph of even girth $2k+2$ with the same asymptotic order and size by constructing the bipartite double cover.

▶ **Lemma 37.** *Let $G=(V,E)$ be a n-node graph with girth $2k+1$, then there is a balanced, bipartite graph $G'=(V',E')$ with girth $2k+2$, $|V'|=2|V|$ and $|E'|=2|E|$.*

Proof. Let $V' := \bigcup_{v \in V}\{v_1, v_2\}$, i.e., for each node $v \in V$ we create two copies. Further, let $E' = \bigcup_{\{u,v\} \in E}\{\{u_1, v_2\}, \{v_1, u_2\}\}$, i.e., for each edge $\{u, v\}$ in E we create two "crossing" edges between the node copies u_1, v_2 and u_2, v_1. Any cycle of G' must form a corresponding cycle in G, by taking the original edge $\{u, v\}$ for each edge $\{u_1, v_2\}$ in that cycle. Thus G' can not have a cycle shorter than $2k+1$. Further, by construction, we have a (balanced) bipartition of G' given by the nodes with index 1 and 2, respectively. Since G' is bipartite, it can not contain an odd cycle, hence the girth is at least $2k+2$. ◀

Combining Lemma 37 with Lemma 35 and the n-node clique which has girth 3 and $\Theta(n^2)$ edges, we obtain the following lemma.

▶ **Lemma 38.** *For $\ell = 4,6,8,12$ there are balanced, bipartite n-node graphs with girth ℓ and $\Theta(n^{1+\frac{2}{\ell-2}})$ edges.*

Note that Lemma 38 this is tight, since for any even $\ell \geq 4$ we obtain the upper bounds $\Omega(n^{1+\frac{2}{\ell-2}})$ in Lemma 34 by plugging in the smaller uneven girth $\ell - 1$. For all other even girths we have to fall back on Lemma 36. Combining it with Lemma 37 gives us the lemma below. Note that we do not apply this lemma for girth 10 as we can get the same asymptotic number of edges for the higher (= better) girth 12 from Lemma 38.

▶ **Lemma 39.** *For any even $\ell \geq 14$ there is a balanced, bipartite n-node graph with girth ℓ and $\Theta(n^{1+\frac{4}{3\ell-10}})$ edges if $\ell \equiv 2 \bmod 4$, or $\Theta(n^{1+\frac{4}{3\ell-12}})$ edges if $\ell \equiv 0 \bmod 4$.*

C Proof of Lemma 9

Proof. We derive a protocol $\mathcal{P}$ that uses (i.e., simulates) algorithm $\mathcal{A}$ in order to solve the two-party communication problem. First, we make a few assumptions about the initial knowledge of both parties in particular about the graph G from the node communication problem, you can think of this information as hard coded into the instructions of $\mathcal{P}$. The important observation is that none of these assumptions give Bob any knowledge about X.

Specifically, assume that Alice is given complete knowledge of the topology G and inputs of all nodes in G (in particular the state of X and the source codes of all nodes specified by $\mathcal{A}$). Bob is given the same for the subgraph induced by $V \setminus A$, which means that the state of X remains unknown to Bob (c.f., Def. 7). To accommodate randomization of $\mathcal{A}$, both are given the same copy of a string of random bits (determined randomly and independently from X) that is sufficiently long to cover all "coin flips" used by any node in the execution of $\mathcal{A}$.

Alice and Bob simulate the following nodes during the simulated execution of algorithm $\mathcal{A}$. For $i \in [h-1]$ let $V_i := \{v \in V \mid \text{hop}(v, A) \leq i\}$ be the set of nodes at hop distance at most i from A. Note that $A \subseteq V_i$ for all i. In round 0 of algorithm $\mathcal{A}$, Alice simulates all nodes in A and Bob simulates all nodes in $V \setminus A$. However, in subsequent rounds $i > 0$, Alice simulates the larger set $A \cup V_i$ and Bob simulates the smaller set $B \cup V \setminus V_i$.

Figuratively speaking, in round i Bob will relinquish control of all nodes that are at hop distance i from set A, to Alice. This means, in each round, every node is simulated *either* by Alice *or* by Bob. We show that each party can simulate their nodes correctly with an induction on i. Initially ($i = 0$), this is true as each party gets the necessary inputs of the nodes they simulate. Say we are at the beginning of round $i > 0$ and the simulation was correct so far. It suffices to show that both parties obtain all messages that are sent (in the $\mathsf{HYBRID}(\infty, \gamma)$ model) to the nodes they currently simulate.

The communication taking place during execution of $\mathcal{A}$ in the $\mathsf{HYBRID}(\infty, \gamma)$ model is simulated as follows. If two nodes that are currently simulated by the *same* party, say Alice, want to communicate, then this can be taken care as part of the internal simulation by Alice. If a node that is currently simulated (w.l.o.g.) by Bob wants to send a message over the *global* network to some node that Alice simulates, then Bob sends that message directly to Alice as part of $\mathcal{P}$, and that message becomes part of the transcript.

Now consider the case where a *local* message is exchanged between some node u simulated by Alice and some node v simulated by Bob. Then in the subsequent round Alice will *always* take control of v, as part of our simulation regime. Thus Alice can continue simulating v correctly as she has all information to simulate all nodes all the time anyway (Alice is initially given all inputs of all nodes). Therefore it is *not* required to exchange any *local* messages across parties for the correct simulation.

After T simulated rounds, Bob, who simulates the set B until the very end (as $T < h$), can derive the state of X from the local information of B with success probability at least p (same as algorithm $\mathcal{A}$). Hence, using the global messages that were exchanged between Alice and Bob during the simulation of algorithm $\mathcal{A}$ we obtain a protocol $\mathcal{P}$ that solves the two party communication problem with probability p. Since total global communication is restricted by $n \cdot \gamma$ bits per round in the $\mathsf{HYBRID}(\infty, \gamma)$ model, Alice sends Bob at most $T \cdot n \cdot \gamma$ bits during the whole simulation. ◀

D Proof of Lemma 17

Proof. Let $U := \{u_1, \ldots, u_k, v\}$ be the vertex cut that separates any s_i from any t_j. The shortest *simple* s_i-t_j-path that crosses U via v has length $w_2 + w_0 + h - 1$ *independently* from x_e (simple implies that a path can not "turn around" and go via u_i).

Consider the shortest s_i-t_j-path that does *not* contain v. In the case $x_e = 1$, i.e., $e = \{u_i, t_j\}$ exists in Γ, this s_i-t_j-path is forced to cross U via u_i and then goes directly to t_j via e, and thus has length $w_2 + w_1 + h - 1$.

Let us analyze the length of the s_i-t_j-path that does *not* contain v for the case $x_e = 0$ (i.e., $e \notin E_\Gamma$). Let G' be the subgraph that corresponds to G after removing each edge $e' \in E$ with $x_{e'} = 0$. Then that s_i-t_j-path has to traverse G' to reach t_j. The sub-path from u_i to t_j in G' has to use at least $(\ell - 1)$ edges, because otherwise $e = \{u_i, t_j\}$ would close a loop of less than ℓ edges in G' (and thus also in G), contradicting the premise that G has girth ℓ. Thus, for $e \notin E_\Gamma$ any s_i-t_j-path that does not contain v has length *at least* $w_2 + (\ell - 1)w_1 + h - 1$.

We sum up the cases. If $x_e = 1$, then the s_i-t_j-path *not* containing v of length $w_2 + w_1 + h - 1$ is shorter than the one via v of length $w_2 + w_0 + h - 1$, since $w_1 < w_0$. If $x_e = 0$, then the s_i-t_j-path via v of length $w_2 + w_0 + h - 1$ is shorter than the one not containing v of length at least $w_2 + (\ell - 1)w_1 + h - 1$ due to $w_0 < (\ell - 1)w_1$. ◀

On Payment Channels in Asynchronous Money Transfer Systems

Oded Naor
Technion, Haifa, Israel

Idit Keidar
Technion, Haifa, Israel

Abstract

Money transfer is an abstraction that realizes the core of cryptocurrencies. It has been shown that, contrary to common belief, money transfer in the presence of Byzantine faults can be implemented in asynchronous networks and does not require consensus. Nonetheless, existing implementations of money transfer still require a quadratic message complexity per payment, making attempts to scale hard. In common blockchains, such as Bitcoin and Ethereum, this cost is mitigated by *payment channels* implemented as a second layer on top of the blockchain allowing to make many off-chain payments between two users who share a channel. Such channels require only on-chain transactions for channel opening and closing, while the intermediate payments are done off-chain with constant message complexity. But payment channels in-use today require synchrony; therefore, they are inadequate for asynchronous money transfer systems.

In this paper, we provide a series of possibility and impossibility results for payment channels in asynchronous money transfer systems. We first prove a quadratic lower bound on the message complexity of on-chain transfers. Then, we explore two types of payment channels, unidirectional and bidirectional. We define them as shared memory abstractions and prove that in certain cases they can be implemented as a second layer on top of an asynchronous money transfer system whereas in other cases it is impossible.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms

Keywords and phrases Blockchains, Asynchrony, Byzantine faults, Payment channels

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.29

Related Version *Full Version*: https://arxiv.org/abs/2202.06693

Funding *Oded Naor*: Oded Naor is grateful to the Azrieli Foundation for the award of an Azrieli Fellowship, and to the Technion Hiroshi Fujiwara Cyber-Security Research Center for providing a research grant.

1 Introduction

The rise of *cryptocurrencies*, such as Bitcoin [33], Ethereum [42], Ripple [2], and many more, has revolutionized the possibility of using decentralized money systems.

In 2019, Guerraoui et al. [20] defined the abstraction of *asset transfer* or *money transfer* capturing the original motivation of Bitcoin. This abstraction is based on a set of known users owning accounts, each account has some initial money, and the users can transfer money between the accounts. It is well-known that deterministic consensus cannot be solved in an asynchronous network [18], meaning that blockchains that rely on consensus require synchrony to work properly. Nonetheless, Guerraoui et al. showed that the asset transfer problem is weaker than consensus [28, 15], i.e., in the Byzantine message-passing model the problem can be solved in an asynchronous network. They provide a concrete implementation of the abstraction in this model using an asynchronous broadcast service.

© Oded Naor and Idit Keidar;
licensed under Creative Commons License CC-BY 4.0

Table 1 Summary of the results.

Abstraction	Operations	Upper bound message complexity	Lower bound message complexity
Asset transfer	transfer read	$O(n^2)$ $O(1)$ [20, 4]	$\Omega(n^2)$ [Theorem 3]
Bidirectional payment channel	open transfer close	Impossible in asynchronous networks [Theorem 5]	
Unidirectional payment channel with source close	open transfer source close target close	Impossible in asynchronous networks [Theorem 7]	
Unidirectional payment channel	open transfer target close	$O(n^2)$ $O(1)$ [Alg. 4] $O(n^2)$	$\Omega(n^2)$ [Lemma 8]

Yet, in this solution, each payment requires a message complexity of $O(n^2)$, where n is the number of processes in the system. If the number of processes grows, this per-payment quadratic message complexity can pose a real challenge in scaling the asset transfer network. In fact, *scalability* is one of the major limiting factors of blockchains and consensus protocols, and extensive research was done to reduce the message complexity [34, 17, 27, 43, 11] in various settings.

A promising approach to scale blockchain payments is by using *payment channels* [36, 29, 38] as a second off-chain layer on top of the blockchain. A payment channel can be *opened* between two blockchain account owners via an on-chain deposit made to fund the channel, after which the two users can *transfer* payments on the channel itself off the blockchain. At any time, one of the users can decide to *close* the channel, after which the current balance in the channel of each of the users is transferred to their on-chain accounts. In this scheme, opening or closing a channel requires a blockchain transaction, which incurs a large message complexity, but all the intermediate payments on the channel require message exchange only between the channel users, freeing the blockchain from these payments and messages.

Payment channels have been actively deployed in central blockchains. For example, Bitcoin's *Lightning Network (LN)* [36] is a highly used payment network, with active channels holding a Bitcoin amount equivalent to hundreds of millions of dollars [40].

One of the downsides of using payment channels such as LN is that the payment channels themselves rely on network synchrony. For example, in LN, suppose Alice and Bob have an open payment channel between them and Alice acts maliciously and tries to steal money by closing the channel at a stale state. LN provides a way for Bob to penalize Alice and confiscate all the money in the channel. But to do so, Bob has to detect Alice's misbehavior on-chain and act within a predetermined time frame, making this method inappropriate with asynchronous users.

In this paper, we explore the possibility of implementing payment channels in asynchronous asset transfer systems. The results of the paper and prior art are summarized in Table 1. We study four abstractions: asset transfer and different types of payment channels.

First, we prove a lower bound on the complexity of asset transfer without channels. We show that no matter what implementation is provided for the asset transfer abstraction, it still requires $\Omega(n^2)$ messages for a single payment to be made by one party and observed by others. This means that the upper bound is tight for the algorithm provided in [20], in which

transferring money has $O(n^2)$ message complexity and reading the balance of an account costs $O(1)$. This fundamental result shows that while the synchrony requirement can be relaxed for asset transfer, each payment still requires a rather large number of messages. This means that second-layer solutions such as payment channels are required for scalability.

Next, we consider payment channel abstractions with operations for opening a channel, transferring money in it, and closing the channel. We first consider a *bidirectional payment channel* as a second layer atop an asset transfer system, where each side of the channel can make payments to the other. This is similar to LN [36], Teechain [29], Sprites [31], and more payment channel proposals. Once we formalize the problem, it is easy to show that synchrony is required, and therefore a bidirectional channel cannot be implemented on top of an asynchronous asset transfer system.

We next look at *unidirectional payment channels*, in which only one user, the channel *source*, can make payments to the other user, the *target*. We differentiate between two types of unidirectional payment channels: If we allow both the source and the target to close the channel (the source and target close operations, respectively), we again show that synchrony is required. Indeed, a previous design of similar channels [39] still requires synchrony.

On the other hand, if we allow only the target to close the channel, we provide a concrete implementation that works in an asynchronous network. In this implementation, the opening and closure of the channel require a payment using the asset transfer system, incurring $O(n^2)$ message complexity, whereas every payment on the channel itself requires a single message from the source user to the target user. We note that once the channel supports a target close the target can claim its transferred funds in the channel, which is not the case when only the source can close the channel.

Finally, we outline an extension of payment channels to payment chains, in which payments are made across multiple channels atomically. Like their 2-party counterparts, a chain payment over unidirectional channels with only target close can be implemented using a technique used in LN. As for other channel types, k-hop chains are equivalent to k-process consensus, i.e., have a consensus number [23] of k.

To conclude, our contributions in this paper are as follows:

- We prove a quadratic message complexity lower bound for asynchronous asset transfer systems.
- We explore payment channels as a key scaling solution for asynchronous asset transfer systems and provide a series of impossibility results for different channel types.
- We provide a concrete possibility result and an implementation for an asynchronous payment channel.

Structure. The rest of the paper is structured as follows: §2 describes the model and preliminaries; §3 details the asset transfer abstraction and proves a lower bound on message complexity; §4 discusses bidirectional payment channels and §5 discusses unidirectional channels; §6 extends the discussion to chain payments; §7 discusses related work; and finally, §8 concludes the paper. Some of the full proofs are deferred to Appendix A.

2 Model and preliminaries

We study a message-passing distributed system that consists of a set $\Pi = \{p_1, \ldots, p_n\}$ of n processes. The processes can interact among themselves by sending messages. An adversary can *corrupt* up to $f < n/3$ processes, where $f \in \Theta(n)$. If not mentioned explicitly, we assume a corrupt process is *Byzantine*, i.e., it can deviate from the prescribed algorithm and act

arbitrarily. Any non-corrupt process is *correct.* A *crash-fail* fault is when a process stops participating in the algorithm. Every two processes share an asynchronous reliable link between them, such that if one correct process sends a message to another correct process, it eventually arrives, and the target can ascertain its source.

We assume the existence of a *Public Key Infrastructure (PKI)*, whereby processes that know a private key can use it to sign messages such that all other processes can verify the signature. The adversary cannot forge a signature if the private key is owned by a correct process. We assume each private key is owned by one process. We further assume *multisignatures* [6], whereby in order to produce a valid signature matching a constant-sized public key, more than one private key is used to sign the message. Note that signing can be sequential.

We study algorithms in the message-passing model that implement abstractions that are defined as *shared-memory objects.* A shared memory object has a set of *operations*, and processes access the object via these operations. Each operation starts with an *invocation* event by a process and ends with a subsequent *response* event. Invocations and responses are discrete events.

An *implementation* or an *algorithm* π of a shared-memory object abstraction is a distributed protocol that defines the behaviors of processes as deterministic state machines, where state transitions are associated with *actions*: sending or receiving messages, and operation invocations or responses. A *global state* of the system is a mapping to states from systems components, i.e., processes and links. An *initial global state* is when all processes are in initial states and there are no messages on the links between the processes. A *run* or an *execution* of an implementation is an alternating series of global states and actions, beginning with some initial global state, such that state transitions occur according to π. We assume that the first action of each process is an invocation of an operation and that it does not invoke another operation before receiving a response for its last invoked operation.

Each execution creates a *history* H that consists of a sequence of matching invocations and responses, each with the assigned process that invoked the operation and the matching responses. A *sub-history* H' *of* H is a subset of H's events. Let $H|_p$ denote the sub-history of H with process p's events.

A history defines a *partial ordering*: *operation* op_1 *precedes* op_2 *in history* H, labeled $op_1 \prec_H op_2$, if op_1's response event happens before op_2's invocation event in H. History H is *sequential* if each invocation, except perhaps the last, is immediately followed by a matching response. An operation is *pending* in history H if it has an invocation event in H but does not have a matching response. A history H' is a *completion of history* H if it is identical to H except for removing zero or more pending operations in H and by adding matching responses for the remaining ones. A shared-memory abstraction is usually defined in terms of a *sequential specification.* A *legal* sequential history is a sequential history that preserves the sequential specification, i.e98., the sequential specification is the set of all legal histories.

The correctness criteria we consider is *Byzantine sequential consistency (BSC).* This allows to extend sequential consistency [3] to runs with Byzantine processes and not only crash-fail and is an adaptation of the definition of Byzantine linearizability [10].

First, we formally define a sequentially consistent history for runs with crash-fail faults.

▶ **Definition 1** (sequentially consistent history)**.** *Let* E *be an execution of an algorithm, and* H *its matching history. Then* H *is sequentially consistent if there exists a completion* $\widetilde{H}$ *of* H *and a legal sequential history* S *such that for every process* p*,* $S|_p = \widetilde{H}|_p$*.*

An algorithm is sequentially consistent if all its histories are sequentially consistent.

For Byzantine sequential consistency (BSC), let $H|_c$ denote the sub-history of H with all of the operations of correct processes. We say a history is BSC if $H|_c$ can be augmented with operations of Byzantine processes such that the completed history is sequentially consistent. Formally:

▶ **Definition 2** (Byzantine sequential consistency (BSC)). *A history H is BSC if there exists a history H' such that $H'|_c = H|_c$, and H' is sequentially consistent.*

Similar to sequential consistency, an algorithm is BSC if all its histories are BSC. We choose BSC as the correctness criterion and not Byzantine linearizability because it simplifies the implementations we provide below. We explain in §5.3 how to change the implementation that we provide to satisfy Byzantine linearizability. The difference between sequential consistency and from linearizability [24] is that linearizability also preserves real-time order, i.e., for any operations op_1, op_2 s.t. $op_1 \prec_{\widetilde{H}} op_2$, then $op_1 \prec_S op_2$.

3 Asset transfer

3.1 Asset transfer abstraction

Let A be an asset transfer abstraction, which is based on the one defined in [20]. A holds a set of accounts. Each account $a \in A$ is defined by some public key, and there is a mapping $owner(a)\colon A \mapsto \Pi$, that matches for each account a the process that can produce a signature corresponding to the public key associated with a. In case of an account b associated with a multisignature, $owner(b)$ is the set of processes whose private keys can produce a signature matching b's public key. The state of each account a is of the form $A(a) \in \mathbb{R}_{\geq 0}$ and represents the balance of the account a. Each account initially holds its initial balance.

A has two operations: The first, $A.read(a)$, returns the balance of account a, i.e., it returns $A(a)$ and can be called by any process. The second is $A.transfer(a, [(b_1, amt_1), \ldots, (b_k, amt_k)])$, which, for every $1 \leq i \leq k$, transfers from account a's balance amt_i and deposits it in b_i. This call succeeds, and returns *success* if it is called by $owner(a)$ and if the account has enough balance to make the transfer, i.e., $A(a) \geq \sum_{i=1}^{k} amt_i$. Otherwise, it returns *fail* and does nothing.

The *transfer* operation is an extension of the one defined in [20] in that it allows to transfer money from one account to multiple accounts, whereas the original work only allows transferring money from one account to another each time.

In this paper, we consider implementations of the asset transfer abstraction that are BSC. We note that the message-passing asset transfer implementation in [20] is based on a reliable broadcast that preserves source order. Their correctness criteria is neither Byzantine linearizability nor BSC, but it ensures that for every transfer operation, there exists a time t such that if a correct process invokes the read operation after t, then it observes the changes made by the transfer. This is a property we use throughout the proofs which are detailed below.

3.2 Message complexity of asset transfer

We begin by proving a quadratic message complexity lower bound on any asset transfer implementation in §3.2.1, and §3.2.2 shows this lower bound is tight by discussing a concrete implementation of an asset transfer that has a quadratic message complexity for a transfer operation and a constant message complexity for a read operation.

3.2.1 Lower bound

We show that if $f \in \Theta(n)$, then there is a quadratic message complexity lower bound in runs in which money is transferred to some account b, and multiple processes read the balance of b. The proof we use for the lower bound follows the technique used in Dolev-Reischuk's lower bound for Byzantine Broadcast [14]. Formally, we prove the following theorem:

▶ **Theorem 3.** *Consider an algorithm that implements the asset transfer abstraction. Then there exists a run with a single transfer invocation and multiple read invocations in which the correct processes send at least* $(f/2)^2$ *messages.*

Proof. Let π be an algorithm that implements asset transfer, and assume by contradiction that in all its runs with a single transfer correct processes send less than $(f/2)^2$ messages. We look at all executions of π with two accounts $a, b \in A$ s.t. $\mathit{owner}(a) = p$ for some process $p \in \Pi$, and initially $A(a) = 1, A(b) = 0$.

Consider first an execution σ_0 in which the adversary, denoted adv_0, corrupts a set V of processes, not including p, such that $|V| = \lceil f/2 \rceil$. Denote the set of remaining correct processes as U. In σ_0, process p calls $A.\mathit{transfer}(a, [(b, 1)])$. By the correctness definition of A, and since p is correct, there exists a time t_0 during the run after which any correct process that invokes $A.\mathit{read}(b)$ returns 1.

The adversary adv_0 causes the corrupt processes in V to simulate the behavior of correct processes that call $A.\mathit{read}(b)$ after t_0, and follow the algorithm except for the following changes: they ignore the first $f/2$ messages they receive from processes in U, and they do not send any message to other processes in V. Note that while t_0 is not known to the processes, we construct the runs from the perspective of a global observer and may invoke read after t_0.

Because correct processes send, in total, less than $(f/2)^2$ messages and corrupt processes do not send messages to other processes in V, then the processes in V together receive less than $(f/2)^2$ messages. Thus, by the pigeonhole principle, there exists at least one process $q \in V$ that receives less than $f/2$ messages. Denote the set of processes that send messages to q as U', and denote $U'' = U \setminus U'$. Note that U' may include process p, and that $|U'| < f/2$.

Next, we construct a run σ_1 with an adversary adv_1 that are the same as σ_0 and adv_0, respectively, except for the following changes: adv_1 corrupts all the processes in $V \setminus \{q\}$, and all the processes in U'. Since $|U'| < f/2$ and $|V| \leq \lceil f/2 \rceil$, the adversary adv_1 corrupts at most f processes in σ_1. adv_1 prevents the corrupt processes from sending any message to q, but causes them to behave correctly towards all other correct processes in U''.

By definition, the behavior of the corrupt processes in σ_1, i.e., the processes in $U' \cup (V \setminus \{q\})$, towards the correct processes in U'' is the same as in σ_0. Since process q simulates a correct process that ignores the first $f/2$ messages in σ_0, its behavior towards the processes in U is identical in both runs as well. Thus, runs σ_0 and σ_1 are indistinguishable for the correct processes in U'', ensuring that they behave the same. Since process q acts in σ_0 like a correct process that does not receive any message, both runs are indistinguishable to it as well.

Nonetheless, process q still has to return a value for its $A.\mathit{read}(b)$ call. Denote the time when the call returns as t_1. If it returns a value different from 1, then we conclude the proof, since it is a violation of the read call specification. Otherwise, we construct a run σ_2 with an adversary adv_2 that are the same as σ_1 and adv_1, respectively, except that there is no transfer invocation and all messages to q are delayed until after t_1. For process q, runs σ_1 and σ_2 are indistinguishable until t_1, therefore it returns 1 for the $A.\mathit{read}(b)$ call, violating the read call specification which should return 0, concluding the proof. ◀

We proved that the lower bound for the message complexity of an asset transfer object is $\Omega(n^2)$, assuming $f \in \Theta(n)$.

3.2.2 Upper bound

In [20], an implementation in the message-passing model for the asset transfer abstraction is provided. It uses a broadcast service defined in [30] that tolerates up to $f < n/3$ Byzantine failures. This broadcast has all the guarantees of reliable broadcast [9] (integrity, agreement, and validity), and also preserves source order, i.e., any two correct processes p_1 and p_2 that deliver messages m and m' broadcast from the same process p_3, do so in the same order. The read operation is computed locally, and the transfer operation consists of a broadcast of a single message.

Several protocols can be used to implement such a source-order broadcast service, including a protocol in [30]. Bracha's reliable broadcast [9] can also be used to implement such a service if each correct process adds a sequence number to each message it broadcasts, and each correct process delivers messages from the same process in the order of the sequence numbers. These protocols have a message complexity of $O(n^2)$ per broadcast, proving that the lower bound message complexity we prove above is tight.

Note that our definition for the *transfer* call of the asset transfer abstraction allows transferring in each invocation money from one account to multiple accounts, while in [20] the transfer call allows a transfer to a single account for each invocation. The implementation in [20] can easily be adjusted to support this change by including in each broadcast message the multiple accounts to which money is transferred.

4 Bidirectional payment channel

We seek to analyze if payment channels that are used in common blockchains as a second layer can also be used similarly on top of an asynchronous asset transfer system. We discuss bidirectional payment channels, in which a channel is opened between two processes by making a transfer on the asset transfer system. After the channel is opened, both processes can make bidirectional payments on the same channel. Either process can close the channel at any time, after which their accounts in the asset transfer system reflect the state of the channel. This abstraction is similar to payment channels in the Lightning Network [36] in Bitcoin [33] and Raiden [38] in Ethereum [42]. We compare our payment channel abstractions to the currently available implementations in the related work in §7.

First, we formally define this abstraction, and then provide an impossibility result, proving it cannot be implemented in asynchronous networks.

4.1 Definition

We define a bidirectional payment channel abstraction as a shared memory object BC. The formal definition is in Specification 1. BC is defined based on the existence of an asset transfer object A. A channel in BC is of the form (a, b), where a, b are accounts in A.

The state of a payment channel $BC(a, b)$ is $\{\mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0}\} \cup \{\perp\}$. The channel $BC(a, b)$ can be *open*, and then $BC(a, b) = (bal_a, bal_b)$, which represents the balances bal_a, bal_b of accounts a, b in the channel (a, b), respectively. If the channel is *closed*, then its state is $BC(a, b) = \perp$.

The set of operations is the following:

- *open.* We do not provide a detailed specification for this call, as we do not require the full specification to prove that there is no implementation for a bidirectional payment channel in the asynchronous message-passing model. Instead, we assume that all channels are open at the beginning of the run with some initial balances.

Specification 1 Bidirectional payment channel abstraction. Operations for process p.

```
Shared Objects:
  A - asset transfer object
  BC - Bidirectional payment channel object
1  Procedure BC.transfer((a, b), amt):
2    if BC(a, b) = ⊥
3      return
4    (bal_a, bal_b) ← BC(a, b)
5    if owner(a) = p ∧ bal_a ≥ amt
6      execute_payment((a, b), −amt, amt)
7    if owner(b) = p ∧ bal_b ≥ amt
8      execute_payment((a, b), amt, −amt)

9  Function execute_payment((a, b), amt_a, amt_b):
10   (bal_a, bal_b) ← BC(a, b)
11   new_bal_a ← bal_a + amt_a
12   new_bal_b ← bal_b + amt_b
13   BC(a, b) ← (new_bal_a, new_bal_b)

14 Procedure BC.close((a, b), bal):
15   if B(a, b) = ⊥
16     return fail
17   (curr_bal_a, curr_bal_b) ← BC(a, b)
18   other_bal = curr_bal_a + curr_bal_b − bal
19   if owner(a) = p
20     if bal ≠ curr_bal_a
21       return fail
22     return execute_close((a, b), bal, other_bal)
23   if owner(b) = p
24     if bal ≠ curr_bal_b
25       return fail
26     return execute_close((a, b), other_bal, bal)
27   return fail

28 Function execute_close((a, b), amt_a, amt_b):
29   A(a) ← A(a) + amt_a
30   A(b) ← A(b) + amt_b
31   BC(a, b) ← ⊥
32   return success
```

- *transfer*$((a, b), amt)$. A payment in channel (a, b) is possible if the channel is open, if the caller process is a valid owner of either a or b, and the caller's balance in the channel is enough to make the payment. Otherwise, it does nothing. After the call ends, the state $BC(a, b)$ is changed to reflect the payment.
- *close*$((a, b), bal)$. This call can be invoked if the caller process is a valid owner of either a or b. The close of the channel is successful if the process that invokes the call does not try to close it with balance bal that is not the amount it has in the channel. Otherwise, the call fails and does nothing. The call transfers to accounts a, b their balances from the channel, and then changes its status to $\perp$. This call returns *success* or *fail* to indicate the outcome of the call.

In this paper, we consider sequentially consistent bidirectional channels.

4.2 Impossibility of a bidirectional payment channel object

We show that implementing a sequentially consistent bidirectional payment channel in the message-passing model requires synchrony. To this end, we solve wait-free consensus among 2 processes with shared registers and an instance of BC. The consensus abstraction has one call, *propose*(v), which is called with some proposal v, and returns a value. The returned value for any process making the call has to be an input of the call from one of the processes, and it has to be the same value for all invocations, regardless of the caller process. We assume in this proof *crash-fail* faults, i.e., a process corrupted by the adversary stops participating in the protocol but does not deviate from it. Since this is an impossibility result and crash-fail faults are weaker than Byzantine faults, it also applies to runs with Byzantine processes.

▶ **Lemma 4.** *Consensus has a wait-free implementation for 2 processes in the read-write shared memory model with an instance of a bidirectional payment channel shared-memory object and shared registers.*

Algorithm 2 Wait-free implementation of consensus among 2 processes using a bidirectional payment channel. Operations for processes $p_1 = owner(a), p_2 = owner(b)$.

```
Shared Objects:
  A - asset transfer object, initially two accounts a, b ∈ A s.t. A(a) = A(b) = 0
  BC - Bidirectional payment channel object, initially BC(a, b) = (1, 1)
  R1, R2 - shared registers with read write calls, initially R1 = R2 = ⊥
// We assume that at the beginning of the run there exists an open payment channel (a, b) s.t.
BC(a, b) = (1, 1)

// Algorithm for process p1:
1  Procedure propose(v):
2    R1.write(v)
3    BC.transfer((a, b), 1)
4    BC.close((a, b), 0)
5    return make_decision()
// Algorithm for process p2:
6  Procedure propose(v):
7    R2.write(v)
8    BC.close((a, b), 1)
9    return make_decision()

// Algorithm for processes p1 and p2:
10 Procedure make_decision():
11   wait until A.read(b) ≠ 0
12   if A.read(b) = 2
13     return R1.read()
14   else
15     return R2.read()
```

Proof. The algorithm for solving consensus among two processes using a bidirectional payment channel object is detailed in Alg. 2. We assume that there are two processes p_1, p_2 with ownership of accounts a, b, respectively, and an open payment channel (a, b) at the beginning of the run with balances $BC(a, b) = (1, 1)$.

Before either of the processes invokes an operation on the payment channel, they write their proposal v in a shared register (Lines 2 and 7). Then, p_1 attempts to make a payment on the channel and then close it, and p_2 tries to close the channel without making or accepting any payment.

Because BC is sequentially consistent, the algorithm ensures that eventually after the channel is closed either the payment from a to b on the channel succeeds or not, and the balance in b's account reflects it, i.e., there exists a time t after which invoking $A.read(b)$ returns either 1 or 2.

If the read call in Alg. 2 returns 2, then the channel was closed by p_1 after it successfully made the payment on the channel. Since before p_1 makes the payment on the channel it writes its proposal to register R_1, then its value is already written by the time the channel is closed, and it is returned by the propose call. If the return value of the read call is 1, then process p_2 closed the channel. Since p_2 closes the channel after it writes its proposal in R_2, then its proposed value is returned.

In either case, when the channel is closed, there is already a proposal written in either R_1 or R_2, i.e., the returned value is an input to the propose call by either process, and both processes return the same value. ◀

Based on the above theorem and FLP [18], we get the following result:

▶ **Theorem 5.** *There does not exist an implementation of the bidirectional payment channel abstraction in the asynchronous message-passing model.*

5 Unidirectional payment channel

After proving that a bidirectional payment channel cannot be implemented in asynchronous networks, we explore another type of payment channel, *unidirectional*. The main difference from bidirectional channels is that unidirectional channels are asymmetric. There is only one

Specification 3 **Unidirectional payment channel abstraction**. Operations for process p.

```
Shared Objects:
  A - asset transfer object, with initial accounts
  B - unidirectional payment channel object
1  Procedure B.open((a, b), amt):
2    if p ≠ owner(a) ∨ B(a, b) ≠ ⊥ ∨ A(a) < amt
3      return fail
4    A(a) ← A(a) − amt
5    B(a, b) ← (amt, 0)
6    return success
7  Procedure B.transfer((a, b), amt):
8    if p ≠ owner(a) ∨ B(a, b) = ⊥
9      return
10   (bal_a, bal_b) ← B(a, b)
11   if bal_a < amt
12     return
13   new_bal_a ← bal_a − amt
14   new_bal_b ← bal_b + amt
15   B(a, b) ← (new_bal_a, new_bal_b)
16 Procedure B.target_close((a, b), bal_b)):
17   if p ≠ owner(b) ∨ B(a, b) = ⊥
18     return fail
19   (curr_bal_a, curr_bal_b) ← B(a, b)
20   if bal_b ≠ curr_bal_b
21     return fail
22   bal_a ← curr_bal_a + curr_bal_b − bal_b
23   return execute_close((a, b), (bal_a, bal_b))
24 Procedure B.source_close((a, b), bal_a):
25   if p ≠ owner(a) ∨ B(a, b) = ⊥
26     return fail
27   (curr_bal_a, curr_bal_b) ← B(a, b)
28   if bal_a ≠ curr_bal_a
29     return fail
30   bal_b ← curr_bal_a + curr_bal_b − bal_a
31   return execute_close((a, b), (bal_a, bal_b))
32 Function execute_close((a, b), (bal_a, bal_b)):
33   A(a) ← A(a) + bal_a
34   A(b) ← A(b) + bal_b
35   B(a, b) ← ⊥
36   return success
```

user, the source, who can open and transfer money in the channel. We show in which cases unidirectional payment channels can be implemented in an asynchronous message-passing network and in which cases they cannot. We begin by formally defining the unidirectional payment channel abstraction.

5.1 Definition

We define a unidirectional payment channel abstraction as a shared memory object B. The formal definition is in Specification 3. B is defined based on the existence of an asset transfer object A. A payment channel in B is of the form (a, b) where a, b are accounts in A.

The state of a payment channel $B(a, b)$ is $\{\mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0}\} \cup \{\perp\}$. Intuitively, a unidirectional payment channel $B(a, b)$ can either be open, and then $B(a, b) = (bal_a, bal_b)$, or closed, and then $B(a, b) = \perp$. The initial state is that all unidirectional payment channels are closed, e.g., for payment channel (a, b), the state is $B(a, b) = \perp$ at the beginning of the run.

The set of operations is the following:

- $open((a, b), amt)$. A process that owns account a can open a unidirectional payment channel with any other account b with amount amt, as long as it has enough balance in A and does not already have an open payment channel with b. The call returns *success* if the channel is opened successfully and *fail* otherwise.
- $transfer((a, b), amt)$. A payment in the payment channel (a, b) is possible if the channel is open, if the caller of the operation a is $owner(a)$ and a has enough balance in the channel to make the payment. This call does not return a response.
- $source_close((a, b), bal_a)$. A source closing of a payment channel (a, b) can be called by $owner(a)$. The call succeeds if the process that invokes the call does not try to close it with balance bal_a that is not the amount it has in the channel. After the call ends, the balances in the channel are transferred to accounts a, b in the asset transfer system. The call returns *success* if the channel is closed successfully and *fail* otherwise.

- *target_close*$((a, b), bal_b)$. A target closure of a payment channel (a, b) is symmetrical to the source_close call, but is invoked by *owner*(b).

We differentiate between two types of unidirectional payment channels, depending on whether the source close call is included in the allowed set of operations of the shared object or not. Note that without source close, the source depends on the target to close the channel to receive its deposit back after the channel is opened. However, for the target to receive its balance from the channel in the asset transfer system, it has to eventually close the channel. When the target closes the channel, the source also receives its respective balance.

We do not consider a channel with source close and without target close, as in this case only the source has operations it can call, and the target relies on the source for all its operations regarding the channel, and cannot receive the funds transferred to it in the channel on-chain unless the source closes the channel. Also, since in this case, only the source has operations it can invoke, this abstraction can be implemented easily in an asynchronous network as it does not require consensus or any interaction at all between the source and target. We also believe this case does not correspond correctly to existing implementations of payment channels as discussed in §7.

5.2 Impossibility of a unidirectional payment channel with source close

We show that a unidirectional payment channel that has the source close operation has a consensus number of at least 2, and therefore cannot be implemented in an asynchronous message passing network. Formally, we prove:

▶ **Lemma 6.** *Consensus has a wait-free implementation for* 2 *processes in the read-write shared memory model with an instance of a unidirectional payment channel with source close shared memory object and shared registers.*

The proof is similar to the proof of Lemma 4 and is deferred to Appendix A. Based on the above lemma and FLP [18], we get the following result:

▶ **Theorem 7.** *There does not exist an implementation of the unidirectional payment channel abstraction with source close in the asynchronous message-passing model.*

5.3 Unidirectional payment channel without source close

Next, we discuss unidirectional payment channel without the source close operation. We first prove a lower bound on the message complexity of an implementation and then prove that this lower bound is tight by providing an implementation for the abstraction in the asynchronous message-passing model.

Lower bound. We prove that any algorithm that implements the unidirectional payment channel specification incurs a combined message complexity for open, transfer, and close of $\Omega(n^2)$. To this end, we prove the following lemma:

▶ **Lemma 8.** *Consider an algorithm that implements the unidirectional payment channel abstraction B, and an asset transfer A. Then there exists a run with B.open, B.transfer, B.target_close, and A.read calls, in which correct processes send at least $(f/2)^2$ messages.*

The full proof is deferred to Appendix A as it shares similar concepts to the lower bound proof of Theorem 3. The intuition is that an *A.transfer* call can be simulated by opening a channel, transferring money in it, and then closing it.

Algorithm 4 Unidirectional payment channel without source_close implementation in the asynchronous message-passing model. Operations for process p.

```
Shared Objects:
  A - asset transfer object
Local variables:
  source[] - a dictionary with the balances of all channels that p is the source, initially ⊥
  target[] - a dictionary with A multisig invocations for all channels p is the target, initially ⊥

   // This call can be invoked by owner(a)
1  Procedure open((a, b), amt) :
2    if owner(a) ≠ p ∨ source[ab] ≠ ⊥ ∨ A.read(a) < amt
3      return fail
4    invoke A.transfer(a, [(ab, amt)])   // ab is multisig
5    create A.transfer(ab, [(a, amt), (b, 0)])
       invocation tx
6    add p's signature to tx   // tx is not a valid
       transaction without owner(b)'s signature
7    send ⟨"open", tx, amt⟩ to owner(b)
8    source[ab] ← (amt, 0)
9    return success

   // This message is received by owner(b)
10 Upon receiving ⟨"open", tx, amt⟩ and A.read(ab) =
     amt:
11   let tx be A.transfer(ab, [(a, amt), (b, 0)]) invocation
12   if owner(b) ≠ p ∨ ¬validate(tx, a) ∨ target[ab] ≠ ⊥
13     return
14   target[ab] ← tx

   // This call can be invoked by owner(a)
15 Procedure transfer((a, b), amt):
16   if owner(a) ≠ p ∨ source[ab] = ⊥
17     return
18   (bal_a, bal_b) ← source[ab]
19   if bal_a < amt
20     return
21   (new_bal_a, new_bal_b) ← (bal_a − amt, bal_b + amt)
22   create A.transfer(ab, [(a, new_bal_a), (b, new_bal_b)])
     invocation tx
23   add p's signature to tx
24   send ⟨"transfer", tx, amt⟩ to owner(b)
25   source[ab] ← (new_bal_a, new_bal_b)

   // This message is received by owner(b)
26 Upon receiving ⟨"transfer", tx, amt⟩:
27   let tx be A.transfer(ab, [(a, bal_a), (b, bal_b)]) invoca-
       tion
28   if owner(b) ≠ p ∨ ¬validate(tx, a) ∨ target[ab] = ⊥
29     return
30   get A.transfer(ab, [(a, c_bal_a), (b, c_bal_b)]) invoca-
       tion from target[ab]          // The currently
                                       stored transaction
31   if c_bal_a ≠ bal_a − amt ∨ c_bal_b ≠ bal_b + amt
32     return
33   target[ab] ← tx                 // store new transaction

   // This call can be invoked by owner(b)
34 Procedure target_close((a, b), bal_b):
35   if owner(b) ≠ p ∨ target[ab] = ⊥
36     return fail
37   get A.transfer(ab, [(a, curr_bal_a), (b, curr_bal_b)])
       transaction tx from target[ab]
38   if bal_b ≠ curr_bal_b
39     return fail
40   add p's signature to tx    // complete the multisig
41   invoke tx      // invoke A with closing transaction
42   target[ab] ← ⊥
43   send ⟨"close", (a, b)⟩ to owner(a)
44   return success

   // This message is received by owner(a)
45 Upon receiving ⟨"close", (a, b)⟩ and A.read(ab) = 0:
46   source[ab] ← ⊥

47 Function validate(tx, a):
48   return tx is a valid invocation of A and it contains
       owner(a)'s signature
```

Upper bound. We provide an algorithm in the asynchronous message-passing model that implements a unidirectional payment channel without source close. The algorithm assumes an asset transfer system A, implemented as in [20], and discussed in §3.2.2. The algorithm is detailed in Alg. 4. We denote an account name with a string $c_1c_2 \cdots c_k \in A$ to refer to an account with a public key that is a k-of-k multisignature of $\{owner(c_1), \ldots, owner(c_k)\}$. For example, to sign an invocation of the asset transfer object A of account ab, like transferring money from ab to another account, both $owner(a)$ and $owner(b)$ need to sign the message with their respective private keys before the call can be invoked. The transfer call with an appropriate multisignature can be invoked by any process, in particular, the last process to sign the invocation and complete the signature. When the algorithm mentions that a process creates an $A.transfer$ invocation, e.g., in lines 5 and 22, it does not mean the process invokes the transfer operation, but rather that it adds its signature to a multisignature message allowing an invocation of A's operation. Any invocation of $A.transfer$ call is explicitly mentioned (lines 4, 40). We further assume FIFO order on messages sent between every two processes. This can be easily implemented with sequence numbers.

We explain below the implementation details of the algorithm for each of the operations:

- Open. The open procedure of a channel (a, b) (Alg. 4) requires $p_1 = owner(a)$ to make an initial deposit by invoking the transfer method of A from account a to a multisignature account ab (Alg. 4). After the transfer is completed, p_1 creates a transaction tx that transfers the deposit back to its account and 0 to $p_2 = owner(b)$ and sends it to p_2 (Alg. 4). Note that at this stage, process p_1 cannot invoke A with tx since it requires a multisignature, but when p_2 receives it, it can add its signature as well and then invoke A with the tx.
  When p_2 receives tx, this transaction message it also waits for the balance in account ab to reflect the deposit (Alg. 4) to ensure the money was deposited in account ab using the asset transfer system, after which it considers the account as open.
- Transfer. When p_1 wants to transfer money in an open channel (a, b) (Alg. 4) it creates a transaction tx which is an $A.transfer$ invocation transferring money from the multisignature account ab to accounts a and b with the last balance of the channel after the payment. E.g., if the balance of the channel is $(10, 1)$, and p_1 wants to make a payment of 1 on the channel, it creates transaction tx required to invoke $A.transfer(ab, [(a, 9), (b, 2)])$, which transfers 9 money units to p_1 and 2 to p_2. Then p_1 adds its signature to tx (Alg. 4), and sends it to p_2, which stores it. Note that p_1 cannot invoke A with tx since it is still missing p_2's signature. Thus, making a payment on the channel simply requires one message from the source user to the target user containing tx, and multiple payments can be made on the channel without invoking A's transfer call.
- Close. When p_2 wants to close the channel (a, b) (Alg. 4), it takes the last transaction of account ab it received from p_1 and adds its signature to it (Alg. 4). p_2's signature completes the multisignature, making it a valid transaction, and allowing p_2 to use it to invoke A's transfer operation (Alg. 4). Process p_2 notifies p_1 that it closed the channel, after which p_1 considers the channel closed. After the channel is closed, p_1 can reopen it with a new call of the open operation.

Thus, opening and closing of the channel requires invoking a single $A.transfer$ operation, which incurs $O(n^2)$ messages because of the broadcast, but transferring money on the channel itself requires only one message per transfer.

Correctness. We prove below that the implementation (Alg. 4) is Byzantine sequentially consistent (BSC) with respect to the sequential specification (Specification 3).

▶ **Definition 9.** *Let E be an execution of Alg. 4 and H its matching history. Let $\widetilde{H}$ be a completion of H by removing any pending open and close calls that did not reach A's transfer call invocation (Lines 4 and 41, respectively), and let $\widetilde{H}|_c$ be $\widetilde{H}$'s history with the operations of correct processes.*

Define H' as an augmentation of $\widetilde{H}|_c$ as follows: For any correct process $q = owner(b)$ that invokes a successful $B.target_close((a, b), bal_b)$ s.t. process $p = owner(a)$ is a Byzantine process, we add before the target close call the following two invocations to H' by p:

- *$B.open((a, b), bal_b)$ with an account a s.t. $A(a) \geq bal_b$. Since account a has enough money to open the channel, the open call succeeds.*
- *$B.transfer((a, b), bal_b)$ which is invoked immediately after the previous open call returns.*

The two added Byzantine calls ensure that when q invokes the close operation, it succeeds. Next, we construct a linearization of H'.

▶ **Definition 10.** *Let H' be the augmented history of Alg. 4 as defined in Definition 9. Let E' be a linearization of H' by defining the following linearization points:*

- *Any open or close call that fails is linearized immediately after its invocation.*
- *A transfer call that returns because of the if statements (Lines 16, 19) is linearized immediately after its invocation.*

- *Any successful open*$((a, b), amt)$ *s.t.* $q = owner(b)$ *is a correct process, then it linearizes after* q *reaches Alg. 4. If* q *is Byzantine, the call linearizes when it ends.*
- *Any transfer*$((a, b), amt)$ *that reaches Alg. 4 s.t.* $q = owner(b)$ *is a correct process, then it linearizes after* q *reaches Alg. 4. If* q *is Byzantine, the call linearizes when it ends.*
- *Any successful target_close*$((a, b), bal_b)$ *s.t.* $p = owner(a)$ *is a correct process, then it linearizes after* p *reaches Alg. 4. If* p *is Byzantine, the call linearizes when it ends.*

The open and transfer calls change the state of the channel. By the sequential specification, the target can call target close with this new state. Therefore, the linearization point of these calls occur after the target receives the message informing it of the new state. Regarding the close call linearization point: the source can only reopen a channel after it learns that the channel has been closed, and therefore the linearization point is when the source receives the information of the closure and verifies it on-chain.

Next, we provide below the lemmas showing that the linearization E' satisfies the sequential specification. We defer the proofs of these lemmas to Appendix A.

▶ **Lemma 11.** *A B.open call for channel* (a, b) *succeeds only if the channel is closed when the call is invoked.*

▶ **Lemma 12.** *For any B.transfer call for channel* (a, b) *there is a preceding open call for the channel in* H'*.*

▶ **Lemma 13.** *For any successful B.target_close*$((a, b), bal_b)$ *call in* H' *there is a preceding B.open*$((a, b), amt)$ *call for channel* (a, b) *s.t.* $amt \geq bal_b$*, followed by a B.transfer call that changes the state of the channel to* (bal_a, bal_b) *for* $bal_a = amt - bal_b$*.*

▶ **Lemma 14.** *In an infinite execution of Alg. 4 every call invoked by a correct process eventually returns.*

Thus, we can conclude the following result from the lemmas above:

▶ **Theorem 15.** *Alg. 4 implements a Byzantine sequentially consistent unidirectional payment channel without source close abstraction.*

Changing the algorithm to be Byzantine linearizable. The algorithm can be changed to be Byzantine linearizable by having each message answered with an ack message. E.g., after the open message is received in Alg. 4, the process sends an ack message back to the original sender. The linearization point then is when the ack message is received. This change requires sending more messages as part of the algorithm and also extends the latency, but this change does not affect the overall asymptotic message complexity. This is also the reason why we choose BSC as the correctness criterion, not Byzantine linearizability.

6 Chain payments

We can extend the discussion of payment channels to chain payments. A chain payment system allows making payments off-chain between users who do not share a direct payment channel between them but do share a route through intermediate users. For example, suppose that Alice wants to make a payment to Bob, but she does not share a direct payment channel with him. Rather, she has a channel with Charlie and Charlie has a channel with Bob. A chain payment allows using the route from Alice to Bob via Charlie to make the payment on all channels atomically. Chain payments are used extensively in Lightning Network [36].

The intuitive way to define a chain payment abstraction is with a single operation that makes a payment through a chain $((a_1, a_2), (a_2, a_3), \ldots, (a_{k-1}, a_k))$, and the outcome of the payment affects all channels of the chain in an atomic manner. For example, suppose that the balances of the above channels of the chain are $(bal_1, bal_2), \ldots, (bal_{k-1}, bal_k)$, respectively, and a payment of *amt* is made via the chain. Then, after the linearization point, the balances of the channels are $(bal_1 - amt, bal_2 + amt), \ldots, (bal_{k-1} - amt, bal_k + amt)$, respectively. In this case, assuming that the channels are bidirectional or unidirectional with source close, it can be proven that the consensus number [23] of such chain payment object is k, meaning this object can be used to solve consensus between k processes, in a similar manner to the 2-consensus we prove in this paper for these channel types.

We also note that even if the channels of the chain are unidirectional without source close, implementing a chain payment is not intuitive and straightforward in asynchronous networks. A possible solution is to adopt the use of *Hash Time-Locked Contracts (HTLCs)* [13, 25] which are in use in the Lightning Network for chain payments. An HTLC is a special conditioned payment between two users Alice and Bob. An HTLC allows Alice to make a conditioned payment to Bob that includes some timeout Δ and a hash value y. Bob can receive the payment if he exposes on-chain a value x whose hash value is y before Δ elapses. We can equip the asset transfer system with a similar Hash Time Contract operation, without the timeout component. That is, Bob receives the payment if he exposes x without any timeout assumptions. With this, we can implement chain payments in asynchronous networks based on the unidirectional channels without source close we presented in Alg. 4. We leave as open future work to formalize these ideas and explore other possible asynchronous implementations of chain payments.

7 Related work

A few works predating the blockchain era [35, 22], identified that a transfer-like system can be implemented under asynchrony. The first definition of the asset transfer abstraction is due to Guerraoui et al. [20]. Subsequentially, Auvolat et al. [4] provide a weaker specification for asset transfer and detail an implementation that uses a broadcast service that guarantees FIFO order between every two processes. Astro [12] implements and empirically evaluates an asynchronous payment system.

Solving Byzantine consensus in an asynchronous network is possible by using randomization to circumvent the FLP [18] result. Earlier protocols such as [7, 37] have exponential message complexity. Later protocols such as [32, 1, 21, 27] improve the message complexity in various settings, but they are not deterministic, and therefore their performance can only be measured in the expected case.

There is also extensive research done on scaling cryptocurrencies using payment channels, including the Lightning Network [36], Teechain [29], Bolt [19], Sprites [31], Perun [16], Duplex Micropayment Channels [13], Raiden [38], and more. In Ethereum [42], there are multiple second-layer networks like Arbitrum [26], StarkNet [8], and Optimism [41]. All these works assume an underlying synchronous network to function correctly. E.g., Lightning Network uses a penalizing mechanism where one side of the channel can confiscate the other side's balance in the channel if it misbehaves and tries to close the channel at a stale state. But for this mechanism to work correctly, the penalizing party has to place a transaction on-chain within a certain time period, thus requiring a synchronous network. Brick [5] is a payment channel that preserves safety and liveness in asynchrony, but to do so requires a rather complex third-party entities (referred to as wardens) which validate channel transactions before they can be placed on-chain. This work also assumes rational (and not Byzantine) behavior of the wardens.

Spilman [39] proposed in 2013 a unidirectional payment channel implementation that shares similar concepts to Alg. 4. Spilman's design has a timeout for each channel. The target of the channel has to close the channel before the timeout passes, otherwise, the source side can refund its initial deposit in the channel. Because of the timeout, this design relies on network synchrony. To the best of our knowledge, our implementation of a unidirectional channel without source close is the first that works on an underlying asynchronous network without requiring any third-party assistance to operate the channel, and in the presence of Byzantine processes.

We are also the first to show a quadratic lower bound for payments in asset transfer systems, as well as to explore second-layer payment channels as a scaling solution in asynchrony.

8 Conclusion

In this paper we presented the possibility of using payment channels in asynchronous asset transfer systems as a scaling solution. We showed that an asset transfer system requires a quadratic message complexity per payment. Then, we showed a series of possibility and impossibility results regarding payment channels as a scaling solution.

References

1 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 337–346, 2019.

2 Frederik Armknecht, Ghassan O Karame, Avikarsha Mandal, Franck Youssef, and Erik Zenner. Ripple: Overview and outlook. In *International Conference on Trust and Trustworthy Computing*, pages 163–180. Springer, 2015.

3 Hagit Attiya and Jennifer L Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems (TOCS)*, 12(2):91–122, 1994.

4 Alex Auvolat, Davide Frey, Michel Raynal, and François Taïani. Money transfer made simple: a specification, a generic algorithm, and its proof. *Bull. EATCS*, 132, 2020. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/629.

5 Zeta Avarikioti, Eleftherios Kokoris-Kogias, Roger Wattenhofer, and Dionysis Zindros. Brick: Asynchronous incentive-compatible payment channels. In *International Conference on Financial Cryptography and Data Security*, pages 209–230. Springer, 2021.

6 Mihir Bellare and Gregory Neven. Identity-based multi-signatures from rsa. In *Cryptographers' Track at the RSA Conference*, pages 145–162. Springer, 2007.

7 Shai Ben-David, Allan Borodin, Richard Karp, Gabor Tardos, and Avi Wigderson. On the power of randomization in on-line algorithms. *Algorithmica*, 11(1):2–14, 1994.

8 Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol. ePrint Arch.*, 2018:46, 2018.

9 Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

10 Shir Cohen and Idit Keidar. Tame the Wild with Byzantine Linearizability: Reliable Broadcast, Snapshots, and Asset Transfer. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DISC.2021.18.

11 Shir Cohen, Idit Keidar, and Alexander Spiegelman. Not a coincidence: Sub-quadratic asynchronous byzantine agreement whp. In *34th International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

12 Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 26–38. IEEE, 2020.

13 Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Symposium on Self-Stabilizing Systems*, pages 3–18. Springer, 2015.

14 Danny Dolev and Ruediger Reischuk. Bounds on information exchange for byzantine agreement. In *Proceedings of the First ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '82, pages 132–140, New York, NY, USA, 1982. Association for Computing Machinery.

15 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

16 Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. Perun: Virtual payment channels over cryptographic currencies. *IACR Cryptol. ePrint Arch.*, 2017:635, 2017.

17 Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *13th USENIX symposium on networked systems design and implementation (NSDI 16)*, pages 45–59, 2016.

18 Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

19 Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 473–489, 2017.

20 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 307–316, 2019.

21 Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: Faster asynchronous bft protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 803–818, 2020.

22 Saurabh Gupta. *A non-consensus based decentralized financial transaction processing model with support for efficient auditing.* Arizona State University, 2016.

23 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

24 Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

25 HTLC. Hash time locked contracts. Accessed: 2022-02-05. URL: https://en.bitcoin.it/wiki/Hash_Time_Locked_Contracts.

26 Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium 18*, pages 1353–1370, 2018.

27 Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 165–175, 2021.

28 Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

29 Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. Teechain: a secure payment network with asynchronous blockchain access. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 63–79, 2019.

30 Dahlia Malkhi and Michael Reiter. A high-throughput secure reliable multicast protocol. *Journal of Computer Security*, 5(2):113–127, 1997.

31 Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. Sprites: Payment channels that go faster than lightning. *CoRR, abs/1702.05812*, 2017. arXiv:1702.05812.

32 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 31–42, 2016.

33 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

34 Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear byzantine smr. In *34th International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

35 Fernando Pedone and André Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2):97–107, 2002.

36 Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

37 Michael O Rabin. Randomized byzantine generals. In *24th annual symposium on foundations of computer science (sfcs 1983)*, pages 403–409. IEEE, 1983.

38 Raiden. Raiden network, 2020. Accessed: 2022-02-05. URL: https://raiden.network/.

39 Jeremy Spilman. Anti dos for tx replacement, April 2013. URL: https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2013-April/002433.html.

40 Lightning Network Statistics. Real-time lightning network statistics. Accessed: 2022-02-01. URL: https://1ml.com/statistics.

41 Optimism website. Accessed: 2022-02-01. URL: https://www.optimism.io/.

42 Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

43 Maofan Yin, Dahlia Malkhi, MK Reiter and, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *38th ACM symposium on Principles of Distributed Computing (PODC'19)*, 2019.

A Correctness proofs

In this section we provide the full deferred proofs from the paper.

▶ **Lemma 6.** *Consensus has a wait-free implementation for* 2 *processes in the read-write shared memory model with an instance of a unidirectional payment channel with source close shared memory object and shared registers.*

Proof. In Alg. 2, only process p_1 transfers money to p_2 using the channel, and they both attempt to close the channel: p_1 after the payment is made, and p_2 without accepting the payment. Thus, changing the close call in Alg. 2 to $B.source_close$ and the call in Alg. 2 to $B.target_close$ yields a consensus algorithm among 2 processes using a unidirectional payment channel with source and target close operations. ◀

▶ **Lemma 8.** *Consider an algorithm that implements the unidirectional payment channel abstraction B, and an asset transfer A. Then there exists a run with $B.open$, $B.transfer$, $B.target_close$, and $A.read$ calls, in which correct processes send at least $(f/2)^2$ messages.*

Proof. We can simulate an $A.transfer$ call between two accounts a, b with initial balances $1, 0$, respectively. This is done by having $owner(a)$ call $B.open((a, b), 1)$, followed $B.transfer((a, b), 1)$, and lastly, having $owner(b)$ call $B.target_close((a, b), 1)$. If accounts a, b are owned by the same process p, we can construct exactly the the same runs used in the proof of Theorem 3 in order to prove this lemma. The only change is that we replace the $A.transfer$ call invoked by p in the original proof with the three calls mentioned above. ◀

Next, we provide the full proofs of the correctness of the unidirectional channel without source close implementation in §5.3. We prove that E', the linearization of H', satisfies the sequential specification of a unidirectional payment channel without source close. We assume that for channel (a, b), $p = owner(a), q = owner(b)$.

▶ **Lemma 11.** *A $B.open$ call for channel (a, b) succeeds only if the channel is closed when the call is invoked.*

Proof. Immediate from the algorithm. A channel (a, b) opening fails if $source[ab] = \bot$ (Alg. 4). This is the case for all channels at the beginning of the run, or if the channel was previously closed successfully (Alg. 4). ◀

▶ **Lemma 12.** *For any $B.transfer$ call for channel (a, b) there is a preceding open call for the channel in H'.*

Proof. If the transfer call in H' is invoked by a correct process, then from the algorithm $source[ab] \neq \bot$. This is only possible by the algorithm if p invokes an open call for the channel before the transfer invocation. If q is correct, then the open and transfer calls linearize when q receives the messages for the corresponding calls (Lines 10, 24). Since we assume FIFO order on the links between any two processes, then the open call linearizes before the transfer call. If q is Byzantine, then the open and transfer calls linearize immediately after they successfully return.

If p is Byzantine, then the transfer invocation is in H' because there is some close invocation by a correct process q. Before that, there is also a matching open call by p. The open call linearizes before the transfer call. ◀

▶ **Lemma 13.** *For any successful $B.target_close((a, b), bal_b)$ call in H' there is a preceding $B.open((a, b), amt)$ call for channel (a, b) s.t. $amt \geq bal_b$, followed by a $B.transfer$ call that changes the state of the channel to (bal_a, bal_b) for $bal_a = amt - bal_b$.*

Proof. If the close call ends successfully, then q has in $target[ab]$ a valid transaction $A.transfer(ab, [(a, bal_a), (b, bal_b)])$, otherwise, the call fails. Therefore, if p is a correct process, it opens the channel (a, b) with some deposit amt, and transfers in the channel s.t. the balances in the channel change to (bal_a, bal_b). Both the open and transfer are linearized before the close invocation, otherwise, the close call fails. If p is Byzantine, then we add the matching open and transfer invocations to H' which are linearized before the close invocation. ◀

▶ **Lemma 14.** *In an infinite execution of Alg. 4 every call invoked by a correct process eventually returns.*

Proof. In all cases where an open or close calls return *fail* it does so immediately, since it is done prior to any invocation of A. Transfer calls that return due to the if statements (Lines 16, 19) also return immediately.

For the open call, the if condition in the channel open call (Alg. 4) ensures that the process that invokes the call owns account a and that it has enough balance to open the channel. We also assume that a correct process does not invoke a new call before a previous call has a response event. Therefore, the conditions checked during the if statement hold when A's transfer call is invoked, and by the asset transfer specification the call succeeds.

A transfer call that does not invoke any of A's calls, nor does it wait for a reply after it sends the transaction in Alg. 4. Therefore, this call also returns immediately.

A target_close call that returns success invokes A with a transfer call that transfers money from account ab (Alg. 4). To reach this line, the process has to first check if the channel is open, and it has the matching transaction in $target[ab]$ during the if statement of the call. Therefore, invoking A will eventually succeed by the asset transfer specification, and the target_close call returns successfully. ◀

The Space Complexity of Scannable Objects with Bounded Components

Sean Ovens
University of Toronto, Canada

Abstract

A fundamental task in the asynchronous shared memory model is obtaining a consistent view of a collection of shared objects while they are being modified concurrently by other processes. A scannable object addresses this problem. A scannable object is a sequence of readable objects called components, each of which can be accessed independently. It also supports the *Scan* operation, which simultaneously reads all of the components of the object. In this paper, we consider the space complexity of an n-process, k-component scannable object implementation from objects with bounded domain sizes. If the value of each component can change only a finite number of times, then there is a simple lock-free implementation from k objects. However, more objects are needed if each component is fully reusable, i.e. for every pair of values v, v', there is a sequence of operations that changes the value of the component from v to v'.

We considered the special case of scannable binary objects, where each component has domain $\{0, 1\}$, in PODC 2021. Here, we present upper and lower bounds on the space complexity of any n-process implementation of a scannable object O with k fully reusable components from an arbitrary set of objects with bounded domain sizes. We construct a lock-free implementation from k objects of the same types as the components of O along with $\lceil \frac{n}{b} \rceil$ objects with domain size 2^b. By weakening the progress condition to obstruction-freedom, we construct an implementation from k objects of the same types as the components of O along with $\lceil \frac{n}{b-1} \rceil$ objects with domain size b.

When the domain size of each component and each object used to implement O is equal to b and $n \leq b^k - bk + k$, we prove that $\frac{1}{2} \cdot (k + \frac{n-1}{b} - \log_b n)$ objects are required. This asymptotically matches our obstruction-free upper bound. When $n > b^k - bk + k$, we prove that $\frac{1}{2} \cdot (b^{k-1} - \frac{(b-1)k+1}{b})$ objects are required. We also present a lower bound on the number of objects needed when the domain sizes of the components and the objects used by the implementation are arbitrary and finite.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases space complexity, lower bound, shared memory, snapshot object

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.30

Funding Support is gratefully acknowledged from the Natural Sciences and Engineering Research Council of Canada under grant RGPIN-2020-04178 and the Ontario Graduate Scholarship (OGS) Program.

Acknowledgements I thank my advisor, Faith Ellen, for the many helpful discussions and proofreading throughout this project. I also thank the anonymous reviewers for their comments.

1 Introduction

A *scannable object* O consists of a sequence of objects $O[1], \ldots, O[k]$ called *components*, each of which stores a value from some domain and supports *Read* along with some other operations. The *Apply*(i, op) operation applies the operation *op* to $O[i]$, where *op* is an operation supported by $O[i]$. A scannable object also supports the *Scan* operation, which returns a consistent view of $O[1], \ldots, O[k]$ at a point during the operation's execution interval.

© Sean Ovens;
licensed under Creative Commons License CC-BY 4.0

A *snapshot object* [1, 2, 4] is a scannable object whose components support the *Read* and *Write* operations. Snapshot objects were formalized independently by Afek, Attiya, Dolev, Gafni, Merritt, and Shavit [1], Anderson [2], and Aspnes and Herlihy [4]. They have been used to simplify the description of obstruction-free consensus algorithms [10], approximate agreement algorithms [8], and implementations of large classes of objects [4, 16].

It is known that a k-component snapshot object implemented from read/write registers requires at least k registers [13]. There are many known implementations that match this lower bound, but all of them either use objects that are large enough to store the result of a *Scan* [1, 2, 17, 19, 20] or use unbounded sequence numbers [1, 9, 14, 15]. There are also implementations that use significantly more than k base objects [3, 18, 7, 25]. Other implementations use unbounded version lists [24]. Prior to our work, it was not well understood how the number of base objects required to implement a scannable object is related to the domain sizes of the base objects and the components. In this paper, we investigate the space complexity of scannable objects with bounded components that are implemented from objects with bounded domains.

Last year, we considered the space complexity of *scannable binary objects* (i.e. scannable objects whose components have domain $\{0, 1\}$) implemented from objects with domain $\{0, 1\}$. In some circumstances, it is possible to implement a scannable binary object from only k objects. For example, consider a scannable binary object O whose components are test-and-set (TAS) objects. A TAS object supports *Read* and *TAS*, which changes the value of the object to 1 and returns its previous value. There is a simple wait-free implementation of O from k TAS objects $T_1, \ldots, T_k$. The object T_i stores the value of component $O[i]$, an *Apply*(i, *Read*) operation reads T_i, and an *Apply*(i, *TAS*) operation applies *TAS* to T_i. A *Scan* repeatedly collects the values in $T_1, \ldots, T_k$ (i.e. reads them one at a time) until it observes the same sequence of values twice in a row. When this happens, the *Scan* returns this sequence of values. Since the value of each component can change at most once, a *Scan* operation will terminate after performing at most $k + 2$ collects. The implementation is correct because the value of a component cannot change from v to a different value v' and then back to v. Hence, the sequence of values returned by the *Scan* must be the actual value of the scannable object at some point during the execution interval of the *Scan*.

More generally, we gave a lock-free, n-process implementation of any k-component scannable binary object from k objects with the same types as the components of the object along with n binary registers. If the components of the scannable binary object are non-monotonic (i.e. their value can be changed from 0 to 1 and from 1 to 0), we show that more than k objects are required. Specifically, any obstruction-free, n-process implementation of a scannable binary object with k non-monotonic components requires at least $n + k - r - 2$ objects with domain $\{0, 1\}$, where $k \geq 2$ and $2^k - 2^{k-r} < n - 2 \leq 2^k - 2^{k-r-1}$. This lower bound applies to single-updater implementations, where only a single process (called the updater) is allowed to change the value of any component. Since the lower bound applies to obstruction-free and single-updater implementations, it applies to lock-free and wait-free implementations that support multiple updaters as well.

In this paper, we generalize our previous results significantly to obtain new upper and lower bounds on the space complexity of scannable object implementations from objects with arbitrary bounded domain sizes. As discussed with scannable binary objects, a k-component scannable object has a wait-free implementation from k objects with the same domain sizes as the components if each component's value can change only a finite number of times. For example, consider a k-component scannable object O consisting of b-bounded counters. A b-bounded counter supports *Read* and *Inc*$_b$, which increases the value of the counter by 1

if its current value is less than $b-1$ and does nothing otherwise. The scannable object O can be implemented from k b-bounded counters, each of which stores the value of one of the components. A *Scan* repeatedly collects the values of the objects until it obtains the same sequence of values twice in a row. Since the value of each component can increase at most $b-1$ times, a *Scan* will terminate after performing at most $k(b-1)+2$ collects.

We show how our lock-free, n-process implementation of any k-component scannable binary object from $n+k$ objects can be generalized to obtain a lock-free, n-process implementation of a k-component scannable object from k objects with the same domain sizes as the components of the scannable object along with $\lceil \frac{n}{b} \rceil$ objects that have domain size 2^b, for any $b \geq 1$. We also construct an obstruction-free implementation from k objects with the same domain sizes as the components of the scannable object along with $\lceil \frac{n}{b-1} \rceil$ read/write registers that have domain size b. We generalize the notion of non-monotonic binary objects to objects with larger domain sizes: An object is *fully reusable* if, for every pair of values v, v' in its domain, there is a sequence of operations that changes its value from v to v'. This is a natural condition that includes many common objects like registers, compare-and-swap objects, and modulo-b counters. A b-bounded counter is not fully reusable, since there is no sequence of operations that changes its value from 1 to 0, for example.

We show that any obstruction-free, n-process implementation of a scannable object with k fully reusable components that have domain size b requires at least $\frac{1}{2} \cdot (k + \frac{n-1}{b} - \log_b n)$ objects with domain size b when $n \leq b^k - bk + k$. When $n > b^k - bk + k$, we show that $\frac{1}{2} \cdot \left(b^{k-1} - \frac{(b-1)k+1}{b}\right)$ objects with domain size b are required. We also prove a lower bound on the number of objects required by any obstruction-free, n-process implementation of a scannable object with k fully reusable components when the domain sizes of the components and the objects used by the implementation are arbitrary, finite values. Just like our lower bound for scannable binary objects, our lower bound in this paper applies to obstruction-free, single-updater implementations, so it applies to lock-free and wait-free implementations that support multiple updaters as well.

Our original lower bound proof for scannable binary objects involves inductively constructing an unordered set of k-component binary vectors $\{V_1, \ldots, V_\ell\}$ and a configuration C_ℓ, for all $\ell \leq min(n-2, 2^{k-1})$, that satisfy the following property: For any execution α from C_ℓ that does not involve the last ℓ scanners, if the scannable binary object does not contain any of the vectors $V_1, \ldots, V_\ell$ during α, then there is a set of ℓ objects that do not change during α. We show how to obtain an $(n-\ell)$-process implementation of a $(k-1)$-component scannable binary object by discarding these ℓ objects. This can be applied repeatedly until we have a 2-process implementation of a scannable binary object with $k' < k$ components, which we show requires at least $k'+1$ objects.

Our technique in this paper builds on these ideas. However, having objects with domain sizes larger than 2 presents several challenges. First, it is not possible to show that some objects stop changing entirely in certain executions from C_ℓ. Instead, we show that there is a set of forbidden values for each object in certain executions from C_ℓ. Second, since it is not possible to obtain a set of objects that stop changing, a new implementation cannot be obtained by discarding objects. Finally, in this paper we need to construct a sequence of vectors $\langle V_1, \ldots, V_\ell \rangle$ rather than a set of vectors. We will explain how our technique differs from our scannable binary object lower bound in more detail in Section 5.

In Section 2, we briefly survey some other related work. We define our model of computation in Section 3. In Section 4, we present our implementations of scannable objects from objects with bounded domain sizes. We prove our space complexity lower bound in Section 5. Finally, we discuss some possible future research directions in Section 6.

2 Related work

Scannable objects, like snapshot objects, have been used to simplify the description of many distributed algorithms and implementations. Aspnes, Attiya, Censor-Hillel, and Ellen [3] described an implementation of a 2-*component max array*, which is a scannable object that consists of 2 max registers. A *max register* supports *MaxWrite*(x), which changes the value of the object to x if and only if the current value of the max register is less than x, and *MaxRead*, which returns the current value of the max register. The authors used a 2-component max array to implement a limited-use snapshot object whose *Scan* and *Update* operations both have polylogarithmic step complexity. Ellen, Gelashvili, Shavit, and Zhu [14] classified some objects by the number of instances required to solve obstruction-free consensus. In certain cases, they showed that it is possible to solve obstruction-free consensus using a scannable object. For example, obstruction-free consensus can be solved among n processes using a scannable object with $n-1$ components that each support *Read* and *Swap*(v), which changes the value of the component to v and returns its previous value.

Consider a k-component scannable object O whose components have domains $D_1, \ldots, D_k$. Then O has a wait-free, single-updater implementation from one single-writer register with domain $D_1 \times \ldots \times D_k$. A *Scan* simply reads the register. The updater locally stores the current value of O in a variable V. When the updater performs an *Apply*(i, op) operation, it locally applies *op* to V and then writes the resulting vector to the register. There is a known wait-free implementation of a single-writer register with any finite domain size d from d single-writer binary registers [23]. Hence, O has a wait-free, single-updater implementation from $\prod_{i=1}^{k} |D_i|$ single-writer binary registers. Chen and Wei [12] gave an implementation of an s-bit single-writer register from $\Theta(\frac{ns}{t})$ instances of t-bit single-writer registers.

Jayanti [20] defined a generalization of a scannable object called an f*-array*, where f is a function. Like a scannable object, an f-array consists of a sequence of components, each of which has its own domain. The domain of an f-array is the cross product of the domains of its components. The function f maps the domain of the f-array to some arbitrary set of values. An f-array supports a generalization of *Scan*, which we call f*-Scan*, that returns the result of applying f to the value of the object. When f is the identity function, f*-Scan* is the same as *Scan*. Jayanti gives a wait-free implementation of a k-component f-array from k objects of the same types as the components of the f-array along with a single LL/SC object large enough to store the result of an f*-Scan*.

Wei, Ben-David, Blelloch, Fatourou, Ruppert, and Sun [24] described an approach for implementing a scannable object whose components are compare-and-swap objects. Their approach uses a versioned compare-and-swap object to store the value of a component. A versioned compare-and-swap object also stores an unbounded version list, which has a complete history of all the successful *CAS* operations applied to the object. Each element of the version list also stores a timestamp. To *Scan* the scannable object, a process first obtains a new timestamp *ts* and then traverses the version list of each component to find the value with the latest timestamp that does not exceed *ts*.

Ellen, Fatourou, and Ruppert [13] proved that, for all $n > k$, an n-process, k-component snapshot implementation requires at least k registers. Jayanti, Tan, and Toueg [21] proved that $n-1$ registers are required to implement a snapshot object with n components, where each component can be modified by only a single process. Both of these lower bound proofs used covering arguments, which were originally introduced by Burns and Lynch [11].

Covering arguments are a standard technique for proving space complexity lower bounds for implementations that use historyless objects. A *historyless* object can support two kinds of operations: *trivial* operations never change the value of the object, and *historyless* operations

set the object to some fixed value that does not depend on the old value of the object. A set of processes $\mathcal{P}$ *covers* a set of historyless objects $\mathcal{B}$ if $|\mathcal{P}| = |\mathcal{B}|$ and, for every $B \in \mathcal{B}$, there is a process in $\mathcal{P}$ that is poised to apply a nontrivial, historyless operation to B in its next step. If each of the processes in $\mathcal{P}$ takes its next step, then the information in $\mathcal{B}$ is overwritten. In order to prove a lower bound of m on the space complexity of an implementation, it suffices to construct a configuration of the implementation in which a set of processes $\mathcal{P}$ covers a set of objects $\mathcal{B}$ with $|\mathcal{B}| = m$. Therefore, the best space lower bound that can be obtained by a covering argument is n, the number of processes. Hence, to obtain our lower bound, we need to use different techniques. Furthermore, our lower bound applies to implementations that use non-historyless objects.

3 Model

We use a standard asynchronous shared memory model in which n processes communicate using shared *objects*. An object has a *domain* of possible *values*, a set of *invocations* that can be applied to it, and a set of possible *responses* to each invocation. The *sequential specification* of an object O defines, for each value v and each invocation *Inv* of O, the resulting value of O and the response to *Inv* when *Inv* is applied to O.

An object O is *fully reusable* if, for all distinct values v, v' of O, there is a sequence of invocations on O that changes its value from v to v'. An example of a fully reusable object with domain $\{0, \ldots, b-1\}$ supports a single invocation that returns its current value x and then changes its value to $x + 1 \mod b$. An example of an object with the same domain that is not fully reusable supports a single invocation that returns its current value x and then changes its value to $min(b-1, x+1)$. For example, there is no sequence of invocations that would change this object's value from 1 to 0.

In this paper, we implement new objects from a set of *base objects*, which are provided by the system. An *implementation* of an object defines a set of base objects and an algorithm for each process to follow for every invocation of the object. For the sake of clarity, we call the invocations of base objects *primitives*, and we call the invocations of implemented objects *operations*.

A *configuration* of an implementation consists of a value for every base object and a state for every process. We use $value(B, C)$ to denote the value of base object B in configuration C. A *step* by a process consists of a primitive applied to some base object and a response to that primitive, followed by a finite amount of local computation by that process, which may then change its state. In an *initial configuration* of an implementation, no processes have taken steps. An *execution* is an alternating sequence of configurations and steps that begins with a configuration. If an execution is finite, then it ends with a configuration. If C is a configuration and α is a finite execution starting with C, then $C\alpha$ denotes the final configuration of α. If an execution α only contains steps by processes in some set P, then we say α is P-only. If P contains exactly one process p_j, then we say α is p_j-only.

Executions are produced by a *scheduler* that decides the order in which processes take steps and the operations on the implemented object that they perform. In every initial configuration of an implementation, every process is idle. When an idle process p_i is chosen to take a step by the scheduler, the scheduler specifies an instance of an operation to p_i, and then the process takes the first step of its algorithm for that operation. When p_i's algorithm terminates, it returns a response to this operation instance, which is now *complete*, and then p_i becomes idle again. An instance of an operation by p_i is *ongoing* in any configuration that occurs after it is given the operation instance and before p_i returns its response. When

the scheduler chooses a process that is not idle, the process only performs the next step of its algorithm. Thus, a process can have at most one ongoing operation in any configuration. When a process takes a step, the response to the primitive that it applies is determined by the value of the base object to which the primitive is applied along with the sequential specification of that base object. A configuration C is called P-idle, where P is a set of processes, if all of the processes in P are idle in C. If $P = \{p_i\}$, then we say C is p_i-idle.

Two configurations C_1, C_2 are *indistinguishable* to a set of processes P if every process in P has the same state in C_1 and C_2. We use $C_1 \overset{P}{\sim} C_2$ to denote this. Suppose that α_1, α_2 are executions beginning with C_1, C_2, respectively. Then α_1 and α_2 are *indistinguishable* to a set of processes P if $C_1 \overset{P}{\sim} C_2$ and every process in P performs the same sequence of steps (and receives the same responses to each of the primitives applied) in α_1 and α_2. We use $\alpha_1 \overset{P}{\sim} \alpha_2$ to denote this. If γ_1 is a P-only execution from C_1, the base objects accessed by P during γ_1 have the same values in C_1 and C_2, and $C_1 \overset{P}{\sim} C_2$, then there is a P-only execution γ_2 from C_2 such that $\gamma_1 \overset{P}{\sim} \gamma_2$ [6].

An object is *readable* if it supports the *Read* invocation, which returns the value of the object. A *scannable object* O is a sequence of readable objects called *components*. We use $O[i]$ to denote the i-th component of the scannable object O. The value of a k-component scannable object O is a vector in $D_1 \times \ldots \times D_k$, where D_i is the domain of component $O[i]$. The object O supports the invocation $\mathit{Apply}(i, \mathit{op})$, which applies the invocation op to $O[i]$. The object O also supports *Scan*, which reads every component of the object simultaneously.

A *single-updater* implementation of a scannable object allows only one process, called the *updater*, to perform *Apply* invocations. The other processes, called *scanners*, can only perform *Scan* invocations. Note that this is different from a *single-writer* implementation [1, 2, 5], in which process p_i can only perform *Apply* invocations on component i.

An execution α from an initial configuration of an implementation of an object O is *linearizable* if there exists a sequence Π of operation instances and responses that satisfies the following three properties.

(i.) Π contains every complete operation instance in α immediately followed by its response. It also contains some subset of the remaining operation instances in α, each of which is immediately followed by some response.

(ii.) If the response to an operation instance op_1 appears before an operation instance op_2 in α and op_2 is in Π, then op_1 appears before op_2 in Π.

(iii.) Π satisfies the sequential specification of O.

The sequence Π is a *linearization* of α. An implementation is *linearizable* if every execution from every initial configuration of the implementation is linearizable.

If an execution α is linearizable, then every complete operation instance in α can be assigned a *linearization point* at which it appears to take effect. Each linearization point must occur at or after the step containing the operation instance and at or before the step containing its response. Operation instances that are not complete in α may also be assigned linearization points, which must occur at or after the step containing the operation instance. An operation instance that has been assigned a linearization point is said to be *linearized*. If there is a sequence of operation instances and responses in which the linearized operation instances are arranged according to their linearization points, each complete operation instance in α is immediately followed by its response, each operation instance that is not complete in α is immediately followed by some response, and Π satisfies the sequential specification of O, then Π is a linearization of α.

An implementation is called *wait-free* if every operation instance by every process completes within a finite number of steps by that process. An implementation is called *lock-free* if every infinite execution of the implementation contains an infinite number of complete

operation instances. An implementation is called *obstruction-free* if every operation instance by every process completes within a finite number of consecutive steps by that process. Note that every wait-free implementation is also lock-free, and every lock-free implementation is also obstruction-free.

4 Upper bound

In this section, we discuss two scannable object implementations from base objects with bounded domain sizes. Throughout this section, let O be a scannable object with k components. Let $\{p_0, \ldots, p_{n-1}\}$ be the set of processes. First, we argue that our lock-free implementation of a scannable binary object from [22] can be generalized to use fewer base objects with larger domain sizes. We obtain a lock-free implementation of O from k objects with the same types as $O[1], \ldots, O[k]$ along with $\lceil \frac{n}{b} \rceil$ base objects with domain equal to the set of all binary strings of length b, each of which supports *Read* and *Set-bit*(i, v). The *Set-bit*(i, v) invocation changes the i-th bit of the object's value to $v \in \{0, 1\}$. Second, by weakening our progress requirement to obstruction-freedom, we show how to obtain an implementation from base objects with smaller domains. We construct an obstruction-free implementation of O from k objects with the same types as $O[1], \ldots, O[k]$ along with $\lceil \frac{n}{b-1} \rceil$ multi-reader, multi-writer registers with domain size b.

4.1 Lock-free implementation

We presented a lock-free, n-process implementation of a k-component scannable binary object S from k objects $B_1, \ldots, B_k$ with the same types as $S[1], \ldots, S[k]$ along with n binary registers $R_1, \ldots, R_n$ [22]. The base objects $B_1, \ldots, B_k$ are used to store the values of $S[1], \ldots, S[k]$, and the registers $R_1, \ldots, R_n$ are used by scanning processes to detect concurrent *Apply* operations.

To perform an *Apply*(ℓ, op) operation, a process writes 0 to all of the registers $R_1, \ldots, R_n$ and then applies *op* to B_ℓ. To *Scan*, process p_i first collects the values in $B_1, \ldots, B_k$ and then writes 1 to R_i. Then p_i repeatedly collects the values in $B_1, \ldots, B_k, R_i$. When it sees the same sequence of values in $B_1, \ldots, B_k$ and reads the value 1 from R_i n times in a row, process p_i returns the sequence of values it read from $B_1, \ldots, B_k$. If the value of some base object B_ℓ changed since p_i's last collect or $R_i = 0$, then p_i writes 1 to R_i and restarts its sequence of collects.

More generally, there is a lock-free, n-process implementation of O from k objects $B_1, \ldots, B_k$ with the same types as $O[1], \ldots, O[k]$ along with $\lceil \frac{n}{b} \rceil$ base objects $R_1, \ldots, R_{\lceil n/b \rceil}$ with domain equal to the set of all binary strings of length b, each of which supports *Read* and *Set-bit*(i, v). To perform an *Apply*(ℓ, op) operation, a process sets all of the bits of $R_1, \ldots, R_{\lceil n/b \rceil}$ to 0. Then, the process applies *op* to B_ℓ. To *Scan*, process p_i first collects the values in $B_1, \ldots, B_k$ and then applies *Set-bit*$(i \mod b, 1)$ to $R_{\lceil (i+1)/b \rceil}$. The remainder of the implementation is similar to our implementation of a scannable binary object, except that p_i uses the $(i \mod b)$-th bit of $R_{\lceil (i+1)/b \rceil}$ to detect concurrent *Apply* operations.

4.2 Obstruction-free implementation

Implementation 1 is a linearizable, obstruction-free implementation of O. It uses k objects $B_1, \ldots, B_k$ with the same types as the components of O, along with $\lceil \frac{n}{b-1} \rceil$ multi-reader, multi-writer registers $R_1, \ldots, R_{\lceil n/(b-1) \rceil}$ with domain $\{0, \ldots, b-1\}$. The base objects $B_1, \ldots, B_k$ are used to store the values of the components. The registers $R_1, \ldots, R_{\lceil n/(b-1) \rceil}$ are used by scanning processes to determine whether other processes are concurrently taking steps.

Each process p_i stores a pair of constants $j = \lceil (i+1)/(b-1) \rceil$ and $v = \big(i \mod (b-1)\big)+1$. The constant j denotes the index of the register that p_i accesses during its *Scan* operations, and v is the value that p_i writes to that register. Notice that $v > 0$ and process p_i is the only process that can write the value v to the register R_j. Hence, if p_i writes the value v to R_j and then p_i later reads the value v from R_j, then p_i knows no other process has modified R_j since p_i last wrote to it.

Process p_i begins a *Scan* operation by collecting the values in $B_1, \ldots, B_k$ on line 7 and then writing the value v to R_j on line 9. Then process p_i begins repeatedly collecting the values in the base objects $B_1, \ldots, B_k, R_j$ on lines 11-12. Once p_i sees the same sequence of values in $B_1, \ldots, B_k$ and it sees the value v in R_j n times in a row, p_i returns the sequence of values it saw in $B_1, \ldots, B_k$. If p_i sees that the value of some base object B_ℓ has changed since p_i's last collect, or the register R_j contains a value other than v, then p_i enters the block on line 13, writes the value v to R_j, and restarts its sequence of collects. Notice that, unlike in the lock-free implementation, it is possible for a pair of scanning processes who share the same register R_j to repeatedly interrupt each other and prevent progress. Hence, this implementation is not lock-free.

An *Apply*(ℓ, op) operation simply writes 0 to all of the registers $R_1, \ldots, R_{\lceil n/(b-1)\rceil}$ and then applies the operation *op* to B_ℓ.

Implementation 1 A linearizable, obstruction-free implementation of a k-component scannable object from $k + \lceil \frac{n}{b-1} \rceil$ base objects.

```
shared:
   B_1, ..., B_k, each initially 0, where B_ℓ is the
   same type as O[ℓ]
   registers R_1, ..., R_⌈n/(b−1)⌉ with domain
   {0, ..., b − 1}, each initially 0

local constants for process p_i:
   j := ⌈(i + 1)/(b − 1)⌉
   v := (i mod (b − 1)) + 1

1  Apply(ℓ, op) by process p_i:
2     for r ∈ {1, ..., ⌈n/(b − 1)⌉} do
3        Write(R_r, 0)
4     end
5     return op(B_ℓ)

6  Scan by process p_i:
7     S ← collect(B_1, ..., B_k)
8     c ← 0
9     Write(R_j, v)
10    while c < n do
11       S′ ← collect(B_1, ..., B_k)
12       v′ ← Read(R_j)
13       if S ≠ S′ or v ≠ v′ then
14          S ← S′
15          c ← 0
16          Write(R_j, v)
17       else
18          c ← c + 1
19       end
20    end
21    return S
```

A process p_i performing a *Scan* operation aims to perform a collect during which none of the objects $B_1, \ldots, B_k$ are modified. This way, p_i can safely return the sequence of values it returned by this collect. If an *Update* operation writes 0 to R_j before p_i has read R_j for the last time on line 12, then p_i will restart its sequence of collects after it next reads R_j. It is possible that an *Apply* operation finishes setting all of the registers $R_1, \ldots, R_{\lceil n/(b-1)\rceil}$ to 0 just before a scanning process p_i sets R_j to v. In this case, the *Apply* might change one of the base objects $B_1, \ldots, B_k$ during a collect by p_i. However, in Lemma 1, we will argue that, for every complete instance of a *Scan* operation *sc*, at least one of the last n collects performed by *sc* does not overlap with any application of a primitive to $B_1, \ldots, B_k$ on line 5.

▶ **Lemma 1.** *Let α be an execution from the initial configuration of Implementation 1. For any complete instance sc of a* Scan *operation in α, there is at least one collect among the last n collects performed by sc during which no* Apply *operation applies a primitive to any base object $B_1, \ldots, B_k$.*

Proof. Suppose that sc is performed by process p_i. Let cl_1 be the first of the final n collects of $B_1, \ldots, B_k$ performed by p_i during sc, and let cl_n be the last. Process p_i does not enter the block on line 14 between cl_1 and cl_n, as this would cause p_i to perform at least n more collects before returning from sc. Hence, p_i does not write v to R_j after cl_1 begins during sc. Furthermore, immediately after each of the last n collects performed by p_i during sc, process p_i reads the value v from R_j. Since p_i is the only process that can write v to R_j, this implies that no process writes to R_j after cl_1 begins and before cl_n ends (i.e. after p_i reads R_1 during cl_1 and before p_i reads $R_{\lceil n/(b-1) \rceil}$ during cl_n). Every *Apply* operation writes 0 to R_j on line 3 before applying a primitive to one of the base objects $B_1, \ldots, B_k$. Therefore, every *Apply* operation that applies a primitive to a base object $B_1, \ldots, B_k$ during one of the final n collects performed by p_i during sc must have written 0 to R_j before cl_1 began. Hence, at most $n-1$ *Apply* operations apply a primitive to a base object $B_1, \ldots, B_k$ during one of the final n collects of sc. ◀

▶ **Theorem 2.** *Implementation 1 is an obstruction-free, linearizable implementation of O.*

Proof. Consider some execution α of Implementation 1. By Lemma 1, there is at least one collect among the final n collects performed by any complete *Scan* operation sc during which no *Apply* operation applies a primitive to any base object $B_1, \ldots, B_k$. We can linearize sc at the beginning of this collect. All *Apply* operations in sc can be linearized when they apply the primitive on line 5.

By inspection of the code, every complete *Apply* operation applies exactly $\lceil n/(b-1) \rceil + 1$ primitives. A process performing a *Scan* operation by itself will execute at most n iterations of the loop on line 10 before terminating. Hence, Implementation 1 is obstruction-free. ◀

5 Lower bound

In this section, we present a lower bound on the number of objects with bounded domain sizes that are required to implement a scannable object. First, we explain the proof technique that we used to obtain a space complexity lower bound for scannable binary objects, since our proof in this section builds on this technique. A key concept in our technique is the notion of a $\mathcal{W}$-absent execution. Let $\mathcal{I}$ be an obstruction-free, single-updater, n-process implementation of a scannable object O, where process p_0 is the updater and processes $p_1, \ldots, p_{n-1}$ are the scanners. Let $\mathcal{W}$ be some set of values of O. Since $\mathcal{I}$ is a single-updater implementation of O, we note that in any p_0-idle configuration of $\mathcal{I}$, the value of the scannable object O is well-defined. Let α be an execution from some p_0-idle configuration C of $\mathcal{I}$. If p_0 is idle in $C\alpha$, then α is $\mathcal{W}$-absent if, for every p_0-idle configuration C' in α, the value of O in C' is not in the set $\mathcal{W}$. If p_0 is not idle in $C\alpha$, then α is $\mathcal{W}$-absent if $\alpha\alpha'$ is $\mathcal{W}$-absent, where α' is the p_0-only execution from $C\alpha$ in which p_0 finishes its ongoing operation in $C\alpha$. An execution β from $C\alpha$ is called a *$\mathcal{W}$-absent extension* of α if $\alpha\beta$ is $\mathcal{W}$-absent. The following observation is from [22].

▶ **Observation 3.** *Let α be a $\mathcal{W}$-absent execution from some p_0-idle configuration C of $\mathcal{I}$.*

(a) *If sc is an instance of a* Scan *operation in α whose response is also in α, then the response of sc is not equal to any vector in $\mathcal{W}$.*

(b) *Any execution from $C\alpha$ in which only the scanners $p_1, \ldots p_{n-1}$ take steps is a $\mathcal{W}$-absent extension of α.*

We originally considered the case in which O is a scannable binary object and $\mathcal{I}$ only uses binary base objects. We inductively construct, for all $\ell \leq min(n-2, 2^{k-1})$, a configuration C_ℓ and a set of ℓ binary k-component vectors $\{V_1, \ldots, V_\ell\}$, such that, for every $\{V_1, \ldots, V_\ell\}$-absent execution α from C_ℓ in which the last ℓ scanners take no steps, there is a set of ℓ

base objects that do not change during α. All of the vectors $V_1, \ldots, V_\ell$ have a 1 in their k-th component. We show that the ℓ base objects that stop changing can be discarded to obtain an implementation of an $(n-\ell)$-process, $(k-1)$-component scannable binary object. This idea is applied repeatedly until we have a 2-process implementation of a scannable binary object with $k' < k$ components, which we show requires at least $k'+1$ base objects.

When the base objects have larger domain sizes, it is not possible to show that a set of base objects stop changing after certain executions from C_ℓ. Hence, we cannot obtain a new implementation by discarding base objects. Instead, we will show how to construct a set of forbidden values for each base object. More precisely, consider an obstruction-free, n-process, single-updater implementation of a scannable object with k components that have domain sizes $c_1, \ldots, c_k$. We show that, for all $\ell \leq min(n-1, \prod_{y=1}^{k} c_y - \sum_{y=1}^{k} c_y + k - 1)$, there is a sequence of distinct k-component vectors $\langle V_1, \ldots, V_\ell \rangle$, a configuration C_ℓ, and a function $\mathcal{X}_\ell$ that maps each base object to a set of forbidden values such that the following property is satisfied: For any $\{V_1, \ldots, V_\ell\}$-absent execution α from C_ℓ in which the last ℓ scanners take no steps, no base object B_x contains any of its forbidden values $\mathcal{X}_\ell(B_x)$ at any point during α. However, the updater is still able to change the object O to any vector other than $V_1, \ldots, V_\ell$ without using any forbidden value for any base object. This allows us to obtain a lower bound on the number of base objects that are needed by the implementation.

In our construction for the scannable binary object lower bound, the order of the vectors $V_1, \ldots, V_\ell$ does not matter. However, for our construction, the order of the vectors is crucial. In particular, for all $i \in \{1, \ldots, \ell\}$, it is important that every possible value of the scannable object $V' \notin \{V_1, \ldots, V_i\}$ can be reached without changing its value to any of the vectors $V_1, \ldots, V_i$. For example, consider a 2-component scannable object that consists of modulo-3 counters. Consider the sequence of vectors $\big\langle (0,1), (1,0) \big\rangle$. Notice that it is impossible to change the value of this scannable object from $(0,0)$ to $(2,2)$ without first changing its value to either $(0,1)$ or $(1,0)$. Hence, our construction would not work with this particular sequence of vectors.

Throughout the remainder of this section, we consider an obstruction-free, single-updater, n-process implementation $\mathcal{I}$ of a scannable object O with k fully reusable components that have bounded domain sizes. Let $p_0, \ldots, p_{n-1}$ be the processes using $\mathcal{I}$, where p_0 is the updater and $p_1, \ldots, p_{n-1}$ are the scanners.

Let $\mathcal{B}$ be the set of base objects used by the implementation $\mathcal{I}$. Let $B_1, \ldots, B_{|\mathcal{B}|}$ be the base objects in $\mathcal{B}$, and, for all $x \in \{1, \ldots, |\mathcal{B}|\}$, let b_x be the domain size of B_x. Without loss of generality, we assume that the domain of B_x is $\{0, \ldots, b_x - 1\}$. Let $b' = \frac{1}{|\mathcal{B}|} \sum_{x=1}^{|\mathcal{B}|} b_x$ be the average domain size of the base objects in $\mathcal{B}$. We assume that $b_x \geq 2$ for all $x \in \{1, \ldots, |\mathcal{B}|\}$. Thus, $b' \geq 2$.

For all $y \in \{1, \ldots, k\}$, let c_y be the domain size of the y-th component $O[y]$ of the implemented object O. We assume that $c_y \geq 2$ for all $y \in \{1, \ldots, k\}$. Let $h = min\big(n - 1, \prod_{y=1}^{k} c_y - \sum_{y=1}^{k} c_y + k - 1\big)$. We will prove that $|\mathcal{B}| \geq \frac{1}{2} \cdot \big(\sum_{y=1}^{k} \log_{b'} c_y + \frac{h}{b'} - \log_{b'}(h+1)\big)$. In particular, consider the case in which $b_1, \ldots, b_{|\mathcal{B}|}, c_1, \ldots, c_k$ are all equal to b. Our lower bound implies that,

1. if $n \leq b^k - bk + k$, then $|\mathcal{B}| \geq \frac{1}{2} \cdot \big(k + \frac{n-1}{b} - \log_b n\big)$, and
2. if $n > b^k - bk + k$, then $|\mathcal{B}| \geq \frac{1}{2} \cdot \big(b^{k-1} - \frac{(b-1)k+1}{b}\big)$.

We will now show how to obtain the sequence of vectors discussed previously. Without loss of generality, we assume that, for all $y \in \{1, \ldots, k\}$, the domain of component $O[y]$ is $\{0, \ldots, c_y - 1\}$.

For all $y \in \{1, \ldots, k\}$ and all $u \in \{0, \ldots, c_y - 1\}$, define $L_y(u)$ as the length of the shortest sequence of operations that changes the value of $O[y]$ from 0 to u. Since $O[y]$ is fully reusable, $L_y(u)$ is well defined. Define a total order $\prec_y$ as follows: For all $u, v \in \{0, \ldots, c_y - 1\}$, $u \prec_y v$ if and only if either (a) $L_y(u) < L_y(v)$, or (b) $L_y(u) = L_y(v)$ and $u < v$. Consider any shortest sequence of operations σ that changes the value of $O[y]$ from 0 to u, for some $u \in \{0, \ldots, c_y - 1\}$. By definition of $\prec_y$, the sequence of values that $O[y]$ takes during σ does not contain any value $v \in \{0, \ldots, c_y - 1\}$ with $u \prec_y v$.

For all $y \in \{1, \ldots, k\}$ and all $u \in \{1, \ldots, c_y - 1\}$, define $L'_y(u)$ as the length of the shortest sequence of operations that changes the value of $O[y]$ from u to 0. Since $O[y]$ is fully reusable, $L'_y(u)$ is well defined. Define a total order $\prec'_y$ as follows: For all $u, v \in \{1, \ldots, c_y - 1\}$, $u \prec'_y v$ if and only if either (a) $L'_y(u) < L'_y(v)$, or (b) $L'_y(u) = L'_y(v)$ and $u < v$. Consider any shortest sequence of operations σ that changes the value of $O[y]$ from u to 0, for some $u \in \{1, \ldots, c_y - 1\}$. By definition of $\prec'_y$, the sequence of values that $O[y]$ takes during σ does not contain any value $v \in \{1, \ldots, c_y - 1\}$ with $u \prec'_y v$.

Let $\mathcal{U} = \{0, \ldots, c_1 - 1\} \times \ldots \times \{0, \ldots, c_k - 1\}$ be the set of values of the scannable object O. Let $\mathcal{V} \subsetneq \mathcal{U}$ be the set of all vectors in $\mathcal{U}$ that have at least two components with nonzero values. Note that $|\mathcal{V}| = \prod_{y=1}^{k} c_y - \sum_{y=1}^{k} c_y + k - 1$. For all $j \in \{1, \ldots, k-1\}$, define $\mathcal{S}_j \subseteq \mathcal{V}$ as the set of all vectors in $\mathcal{V}$ whose first $j - 1$ components contain the value 0 and whose j-th components contain a nonzero value. Notice that $\mathcal{S}_1, \ldots, \mathcal{S}_{k-1}$ is a partition of the set $\mathcal{V}$.

For all $j \in \{1, \ldots, k-1\}$, let Γ_j be the sequence of all vectors in $\mathcal{S}_j$ ordered first by decreasing lexicographical order of the final $k - j$ components with respect to $\prec_y$, and then in decreasing order by the value in the j-th component with respect to $\prec'_j$. For example, if O consists of $k = 3$ modulo-3 counters, then $2 \prec'_y 1$ and $0 \prec_y 1 \prec_y 2$ for all y. In this case, $\Gamma_1 = \langle [1,2,2], [2,2,2], [1,2,1], \ldots, [2,1,0], [1,0,1], [2,0,1] \rangle$ and $\Gamma_2 = \langle [0,1,2], [0,2,2], [0,1,1], [0,2,1] \rangle$. For all $i \in \{1, \ldots, |\mathcal{V}|\}$, define V_i as the i-th vector in the concatenation of $\Gamma_1, \ldots, \Gamma_{k-1}$. We use $[0, \ldots, 0]$ to denote the k-component all 0 vector.

▶ **Lemma 4.** *For any $i \in \{1, \ldots, |\mathcal{V}|\}$, any $U \in \mathcal{U} - \{V_1, \ldots, V_i\}$, and any p_0-idle configuration C in which the scannable object O contains $[0, \ldots, 0]$, there is a p_0-only, $\{V_1, \ldots, V_i\}$-absent execution λ from C such that $C\lambda$ is p_0-idle and the object O contains U in $C\lambda$.*

Proof. First suppose that $U \in \mathcal{U} - \mathcal{V}$. If $U = [0, \ldots, 0]$, then let λ be the empty execution. Otherwise, let the j-th component of U be a nonzero value. Let λ be some p_0-only execution from C in which p_0 changes the value of $O[j]$ from 0 to $U[j]$. In every p_0-idle configuration of λ, the value of O is a vector in $\mathcal{U} - \mathcal{V}$. Hence, λ is $\{V_1, \ldots, V_i\}$-absent.

Otherwise, $U = V_{i'}$, where $i < i' \leq |\mathcal{V}|$. Suppose that $V_{i'} \in \mathcal{S}_j$, for some $j \in \{1, \ldots, k-1\}$. Let λ_j be the p_0-only execution from C in which p_0 performs a shortest sequence of operations that changes the value of $O[j]$ from 0 to $V_{i'}[j]$. For all $y \in \{j+1, \ldots, k\}$, let λ_y be the p_0-only execution from $C\lambda_j \ldots \lambda_{y-1}$ in which p_0 performs a shortest sequence of operations that changes the value of $O[y]$ from 0 to $V_{i'}[y]$. Note that the sequence of values of $O[y]$ during λ_y are in increasing order with respect to $\prec_y$. Since $V_{i'} \in \mathcal{S}_j$, the first $j - 1$ components of $V_{i'}$ contain the value 0. Hence, in $C\lambda_j \ldots \lambda_k$, the value of O is the vector $V_{i'}$.

During the execution λ_j, the updater p_0 changes the value of $O[j]$ from 0 to $V_{i'}[j]$. All of the other components of O contain 0 throughout λ_j. Hence, the object O only contains vectors in $\mathcal{U} - \mathcal{V}$ during λ_j. Thus, λ_j is $\{V_1, \ldots, V_i\}$-absent.

In the configuration $C\lambda_j$, component $O[j]$ contains the value $V_{i'}[j]$, and component $O[j]$ is not changed during $\lambda_{j+1} \ldots \lambda_k$. Furthermore, the first $j - 1$ components of O contain the value 0 throughout $\lambda_j \ldots \lambda_k$. Hence, the value of the object O in every p_0-idle configuration that appears after $C\lambda_j$ in the execution $\lambda_{j+1} \ldots \lambda_k$ is a vector in $\mathcal{S}_j$. For all $y \in \{j+1, \ldots, k\}$,

every operation that p_0 applies to $O[y]$ increases the value of $O[y]$ with respect to the order $\prec_y$. Hence, the sequence of all values of O are in increasing lexicographical order with respect to $\prec_y$ and the final vector in this sequence is $V_{i'}$. Every vector V that appears before $V_{i'}$ in Γ_j with $V[j] = V_{i'}[j]$ is lexicographically larger than $V_{i'}$ with respect to $\prec_y$. Hence, $\lambda = \lambda_j \dots \lambda_k$ is $\{V_1, \dots, V_i\}$-absent. ◀

▶ **Lemma 5.** *For any $i \in \{1, \dots, |\mathcal{V}|\}$, any $U \in \mathcal{U} - \{V_1, \dots, V_i\}$, and any p_0-idle configuration C in which the scannable object O contains the value U, there is a p_0-only, $\{V_1, \dots, V_i\}$-absent execution τ from C such that $C\tau$ is p_0-idle and the object O contains $[0, \dots, 0]$ in $C\tau$.*

Proof. First suppose that $U \in \mathcal{U} - \mathcal{V}$. If $U = [0, \dots, 0]$, then let τ be the empty execution. Otherwise, let the j-th component of U be a nonzero value. Let τ be a p_0-only execution from C in which p_0 changes the value of $O[j]$ from $U[j]$ to 0. In every p_0-idle configuration of τ, the value of O is a vector in $\mathcal{U} - \mathcal{V}$. Hence, τ is $\{V_1, \dots, V_i\}$-absent.

Otherwise, $U = V_{i'}$, where $i < i' \leq |\mathcal{V}|$. Suppose that $V_{i'} \in \mathcal{S}_j$, for some $j \in \{1, \dots, k-1\}$. Let τ_j be the p_0-only execution from C in which p_0 performs a shortest sequence of operations that changes the value of $O[j]$ from $V_{i'}[j]$ to 0. For all $y \in \{j+1, \dots, k\}$, let τ_y be the p_0-only execution from $C\tau_j \dots \tau_{y-1}$ in which p_0 performs a shortest sequence of operations that changes the value of $O[y]$ from $V_{i'}[j]$ to 0. Since $V_{i'} \in \mathcal{S}_j$, the first $j-1$ components of $V_{i'}$ contain the value 0. Hence, in $C\tau_j \dots \tau_k$, the value of O is $[0, \dots, 0]$.

During the execution τ_j, the updater p_0 changes the value of $O[j]$ from $V_{i'}[j]$ to 0. None of the other components are modified during τ_j. Recall that $\mathcal{S}_j$ is ordered first in decreasing lexicographical order by the last $k-j$ components with respect to $\prec_y$, and then in decreasing order by the j-th component with respect to $\prec'_j$. Every operation by p_0 in τ_j decreases the value of $O[j]$ with respect to $\prec'_j$. In configuration $C\tau_j$, the value of O is a vector Y that contains 0 in its first $j+1$ components. Hence, either $Y \in \mathcal{S}_{j+1} \cup \dots \cup \mathcal{S}_{k-1}$ or $Y \in \mathcal{U} - \mathcal{V}$. Thus, τ_j is $\{V_1, \dots, V_i\}$-absent.

Notice that p_0 does not modify any of the first $j+1$ components of O after the configuration $C\tau_j$ in $\tau_{j+1} \dots \tau_k$. That is, in every configuration of $\tau_{j+1} \dots \tau_k$ after $C\tau_j$, the value of the object is a vector that contains 0 in its first $j+1$ components. Thus, $\tau = \tau_j \dots \tau_k$ is $\{V_1, \dots, V_i\}$-absent. ◀

We will now prove our main technical lemma, which constructs a set of forbidden values for each of the base objects in $\mathcal{B}$. Let C_0 be an initial configuration of $\mathcal{I}$ in which O contains the value $[0, \dots, 0]$. In the following lemma, we use induction to show that, for all $0 \leq \ell \leq h$, there is an execution α_ℓ from C_0 and a function $\mathcal{X}_\ell$ that maps each base object B_x to a proper subset of $\{0, \dots, b_x - 1\}$, where $\mathcal{X}_\ell(B_x)$ represents the set of forbidden values for the base object B_x. The first $n - \ell$ processes are idle and O contains $[0, \dots, 0]$ in the configuration

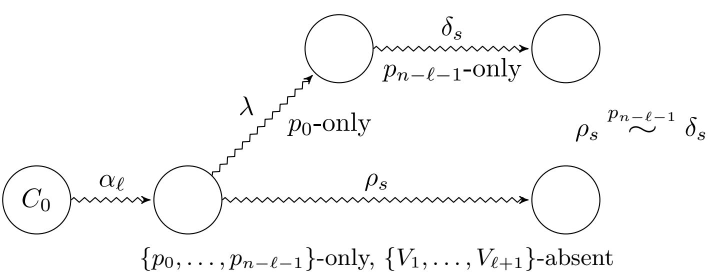


Figure 1 The executions δ_s and ρ_s in the proof of Lemma 6. Notice that δ_s starts from $C_0\alpha_\ell\lambda$ and ρ_s starts from $C_0\alpha_\ell$.

$C_0\alpha_\ell$. Furthermore, the number of forbidden values summed over all the base objects is exactly ℓ. For any $\{p_0, \ldots, p_{n-\ell-1}\}$-only, $\{V_1, \ldots, V_\ell\}$-absent execution from $C_0\alpha_\ell$, none of the forbidden values are used during that execution. To complete the inductive step, we show how to obtain one more forbidden value by stalling the scanner $p_{n-\ell-1}$.

▶ **Lemma 6.** *For all ℓ such that $0 \leq \ell \leq h$, there is an execution α_ℓ from C_0 and a function $\mathcal{X}_\ell$ that maps each base object $B_x \in \mathcal{B}$ to a proper subset of $\{0, \ldots, b_x - 1\}$ such that*
(a) *$C_0\alpha_\ell$ is $\{p_0, \ldots, p_{n-\ell-1}\}$-idle,*
(b) *the object O contains the value $[0, \ldots, 0]$ in $C_0\alpha_\ell$,*
(c) $\sum_{x=1}^{|\mathcal{B}|} |\mathcal{X}_\ell(B_x)| = \ell$,
(d) *for every $\{p_0, \ldots, p_{n-\ell-1}\}$-only, $\{V_1, \ldots, V_\ell\}$-absent execution γ from $C_0\alpha_\ell$ and every $B_x \in \mathcal{B}$, we have* $\text{value}(B_x, C_0\alpha_\ell\gamma) \notin \mathcal{X}_\ell(B_x)$.

Proof. We use induction on ℓ. Let α_0 be the empty execution and let $\mathcal{X}_0(B_x) = \emptyset$ for all $B_x \in \mathcal{B}$. Since no processes have taken any steps in $C_0\alpha_0 = C_0$, part (a) holds. Since all of the components contain the value 0 in the configuration $C_0\alpha_0 = C_0$, this gives us part (b). Since $\mathcal{X}_0(B_x) = \emptyset$ for all $B_x \in \mathcal{B}$, we know that $\sum_{x=1}^{|\mathcal{B}|} \mathcal{X}_0(B_x) = 0$, which gives us part (c) and part (d). This concludes the proof of the base case.

Now let $0 \leq \ell < h$ and suppose the lemma holds for ℓ. Then there exists an execution α_ℓ from C_0 that satisfies parts (a)–(d) of the lemma statement. By Lemma 4 (with $i = \ell$, $U = V_{\ell+1}$, and $C = C_0\alpha_\ell$), there is a p_0-only, $\{V_1, \ldots, V_\ell\}$-absent execution λ from $C_0\alpha_\ell$ such that p_0 is idle in $C_0\alpha_\ell\lambda$ and the object O contains the value $V_{\ell+1}$ in $C_0\alpha_\ell\lambda$.

Process $p_{n-\ell-1}$ is idle in $C_0\alpha_\ell$ by property (a). Since $p_{n-\ell-1}$ takes no steps in λ, it is idle in $C_0\alpha_\ell\lambda$ as well. Let δ be the $p_{n-\ell-1}$-only execution from $C_0\alpha_\ell\lambda$ in which $p_{n-\ell-1}$ does a complete *Scan*. Since the value of O is $V_{\ell+1}$ in $C_0\alpha_\ell\lambda$, the *Scan* operation by $p_{n-\ell-1}$ in δ returns the vector $V_{\ell+1}$. Furthermore, the execution δ is a $\{V_1, \ldots, V_\ell\}$-absent extension of λ by Observation 3 (b). Let r be the number of steps by $p_{n-\ell-1}$ in δ. Define δ_s as the prefix of δ consisting of the first s steps by $p_{n-\ell-1}$. (In particular, δ_0 is empty and $\delta_r = \delta$.)

Let ρ_0 be the empty execution from $C_0\alpha_\ell$. Since $p_{n-\ell-1}$ takes no steps in λ, we know that $C_0\alpha_\ell \overset{p_{n-\ell-1}}{\sim} C_0\alpha_\ell\lambda$. Furthermore, since $p_{n-\ell-1}$ takes no steps in either ρ_0 or δ_0, we know that $\rho_0 \overset{p_{n-\ell-1}}{\sim} \delta_0$. Since O contains the value $[0, \ldots, 0]$ in $C_0\alpha_\ell\rho_0 = C_0\alpha_\ell$ by property (b), the execution ρ_0 is $\{V_1, \ldots, V_{\ell+1}\}$-absent.

Let ρ_r be any $\{p_0, \ldots, p_{n-\ell-1}\}$-only execution from $C_0\alpha_\ell$ such that $\rho_r \overset{p_{n-\ell-1}}{\sim} \delta_r$. Then $p_{n-\ell-1}$'s *Scan* operation in ρ_r returns the vector $V_{\ell+1}$. Hence, by the contrapositive of Observation 3 (a), the execution ρ_r is not $\{V_1, \ldots, V_{\ell+1}\}$-absent.

Let $s \in \{0, \ldots, r-1\}$ be the maximum value such that there is a $\{p_0, \ldots, p_{n-\ell-1}\}$-only, $\{V_1, \ldots, V_{\ell+1}\}$-absent execution ρ_s from $C_0\alpha_\ell$ such that $\rho_s \overset{p_{n-\ell-1}}{\sim} \delta_s$. Then there is no $\{p_0, \ldots, p_{n-\ell-1}\}$-only, $\{V_1, \ldots, V_{\ell+1}\}$-absent extension ρ' of ρ_s such that $\rho_s\rho' \overset{p_{n-\ell-1}}{\sim} \delta_{s+1}$. Suppose that $p_{n-\ell-1}$ is poised to access the base object B_w in $C_0\alpha_\ell\lambda\delta_s$ and $C_0\alpha_\ell\rho_s$. Let d be the last step of δ_{s+1}. If there is a $\{p_0, \ldots, p_{n-\ell-2}\}$-only, $\{V_1, \ldots, V_{\ell+1}\}$-absent extension ρ' of ρ_s such that $\mathit{value}(B_w, C_0\alpha_\ell\rho_s\rho') = \mathit{value}(B_w, C_0\alpha_\ell\delta_s)$, then $\rho_s\rho'd \overset{p_{n-\ell-1}}{\sim} \delta_s d = \delta_{s+1}$. By Observation 3 (b), the execution $\rho_s\rho'd$ is $\{V_1, \ldots, V_{\ell+1}\}$-absent. This contradicts the definition of s. Hence,

$$\mathit{value}(B_w, C_0\alpha_\ell\rho_s\rho') \neq \mathit{value}(B_w, C_0\alpha_\ell\lambda\delta_s) \text{ for every } \{p_0, \ldots, p_{n-\ell-2}\}\text{-only, } \{V_1, \ldots, V_{\ell+1}\}\text{-absent extension } \rho' \text{ of } \rho_s. \tag{1}$$

Let σ_ℓ be the $\{p_0, \ldots, p_{n-\ell-2}\}$-only execution from $C_0\alpha_i\rho_s$ in which the processes $p_0, \ldots, p_{n-\ell-2}$ complete their pending operations in increasing order of their identifiers. Suppose that $\sigma_\ell = \sigma'_\ell\sigma''_\ell$, where σ'_ℓ is the prefix of σ_ℓ that contains all of p_0's steps in σ_ℓ.

Since p_0 does not begin any new *Apply* operations during σ'_ℓ, it is a $\{V_1, \ldots, V_{\ell+1}\}$-absent extension of ρ_s. Furthermore, since σ''_ℓ only contains steps by the scanners $p_1, \ldots, p_{n-1}$, Observation 3 (b) implies that σ''_ℓ is a $\{V_1, \ldots, V_{\ell+1}\}$-absent extension of $\rho_s\sigma'_\ell$. Thus, σ_ℓ is a $\{V_1, \ldots, V_{\ell+1}\}$-absent extension of ρ_s.

Let Y be the value of the object O in configuration $C_0\alpha_\ell\rho_s\sigma_\ell$. Since $\rho_s\sigma_\ell$ is $\{V_1, \ldots, V_{\ell+1}\}$-absent, we know that $Y \in \mathcal{U} - \{V_1, \ldots, V_{\ell+1}\}$. By Lemma 5 (with $i = \ell + 1$, $U = Y$, and $C = C_0\alpha_\ell\rho_s\sigma_\ell$), there exists a p_0-only, $\{V_1, \ldots, V_{\ell+1}\}$-absent execution τ_ℓ from $C_0\alpha_\ell\rho_s\sigma_\ell$ such that p_0 is idle in $C_0\alpha_\ell\rho_s\sigma_\ell\tau_\ell$ and the object O contains $[0, \ldots, 0]$ in $C_0\alpha_\ell\rho_s\sigma_\ell\tau_\ell$.

Let $\alpha_{\ell+1} = \alpha_\ell\rho_s\sigma_\ell\tau_\ell$. In configuration $C_0\alpha_\ell\rho_s\sigma_\ell\tau_\ell = C_0\alpha_{\ell+1}$, the object O contains the value $[0, \ldots, 0]$. This gives us property (b) for $\ell + 1$.

By definition of σ_ℓ, the configuration $C_0\alpha_\ell\rho_s\sigma_\ell$ is $\{p_0, \ldots, p_{n-\ell-2}\}$-idle. Since processes $p_1, \ldots, p_{n-\ell-2}$ take no steps during τ_ℓ, configuration $C_0\alpha_\ell\rho_s\sigma_\ell\tau_\ell = C_0\alpha_{\ell+1}$ is $\{p_1, \ldots, p_{n-\ell-2}\}$-idle. Furthermore, this configuration is p_0-idle by definition of τ_ℓ. This gives us property (a) for $\ell + 1$.

For all $B_x \in \mathcal{B}$, define

$$\mathcal{X}_{\ell+1}(B_x) = \begin{cases} \mathcal{X}_\ell(B_x) \cup \{value(B_x, C_0\alpha_\ell\lambda\delta_s)\} & \text{if } B_x = B_w \\ \mathcal{X}_\ell(B_x) & \text{otherwise.} \end{cases}$$

Recall that $\lambda\delta_s$ is $\{V_1, \ldots, V_\ell\}$-absent. Hence, $value(B_w, C_0\alpha_\ell\lambda\delta_s) \notin \mathcal{X}_\ell(B_w)$ by property (d) for ℓ with $\gamma = \lambda\delta_s$. Thus, we have $|\mathcal{X}_{\ell+1}(B_w)| = |\mathcal{X}_\ell(B_w)| + 1$. Since $\sum_{x=1}^{|\mathcal{B}|} |\mathcal{X}_\ell(B_x)| = \ell$ by property (c) for ℓ, we have $\sum_{x=1}^{|\mathcal{B}|} |\mathcal{X}_{\ell+1}(B_x)| = \ell + 1$. This gives us property (c) for $\ell + 1$.

Let γ' be a $\{p_0, \ldots, p_{n-\ell-2}\}$-only, $\{V_1, \ldots, V_{\ell+1}\}$-absent execution from $C_0\alpha_{\ell+1}$. Then $\rho_s\sigma_\ell\tau_\ell\gamma'$ is a $\{p_0, \ldots, p_{n-\ell-1}\}$-only, $\{V_1, \ldots, V_\ell\}$-absent execution from $C_0\alpha_\ell$. By property (d) for ℓ with $\gamma = \rho_s\sigma_\ell\tau_\ell\gamma'$, for every $B_x \in \mathcal{B}$, we have $value(B_x, C_0\alpha_\ell\rho_s\sigma_\ell\tau_\ell\gamma') \notin \mathcal{X}_\ell(B_x)$. By (1) with $\rho' = \sigma_\ell\tau_\ell\gamma'$, we have $value(B_w, C_0\alpha_\ell\rho_s\sigma_\ell\tau_\ell\gamma') \neq value(B_w, C_0\alpha_\ell\lambda\delta_s)$. This completes the proof of property (d) for $\ell+1$. Hence, by induction, the lemma holds for all $\ell \in \{0, \ldots, h\}$. ◀

We apply Lemma 6 with $\ell = h$ to obtain an execution α_h and h forbidden values for the base objects. We apply Lemma 4 to obtain p_0-only, $\{V_1, \ldots, V_h\}$-absent executions from $C_0\alpha_h$ in which p_0 changes the value of O to the vectors in $\mathcal{U} - \{V_1, \ldots, V_h\}$. None of the forbidden values of any base objects can be used in these executions. This allows us to obtain a lower bound on the number of base objects in $\mathcal{B}$. We provide a sketch of the proof in the following theorem, and complete the proof in Appendix A.

▶ **Theorem 7.** $|\mathcal{B}| \geq \frac{1}{2} \cdot \left(\sum_{y=1}^{k} \log_{b'} c_y + \frac{h}{b'} - \log_{b'}(h+1)\right)$.

Proof sketch. Apply Lemma 6 with $\ell = h$ to obtain an execution α_h and a function $\mathcal{X}_h$ that satisfy (a)–(d). By property (c), we have $\sum_{x=1}^{|\mathcal{B}|} |\mathcal{X}_h(B_x)| = h$. Since $|\mathcal{X}_h(B_x)| \leq b_x - 1$ for all $B_x \in \mathcal{B}$, we have $\sum_{x=1}^{|\mathcal{B}|} (b_x - 1) \geq h$. Hence, $\frac{1}{|\mathcal{B}|} \cdot \sum_{x=1}^{|\mathcal{B}|} (b_x - 1) \geq \frac{h}{|\mathcal{B}|}$. Therefore, $|\mathcal{B}| \geq \frac{h}{b'-1}$.

Let $\mathcal{V}'$ be the set of all values of O except for $V_1, \ldots, V_h$. Then $|\mathcal{V}'| = \prod_{y=1}^{k} c_y \ - h$. By Lemma 6 (b), the object O contains the value $[0, \ldots, 0]$ in $C_0\alpha_h$. For all $V' \in \mathcal{V}'$, there exists a p_0-only, $\{V_1, \ldots, V_h\}$-absent execution $\gamma_{V'}$ from $C_0\alpha_h$ such that p_0 is idle in $C_0\alpha_h\gamma_{V'}$ and O contains V' in $C_0\alpha_h\gamma_{V'}$ by Lemma 4 (with $i = h$, $U = V'$, and $C = C_0\alpha_h$).

Let V'_1, V'_2 be two distinct vectors in $\mathcal{V}'$. Consider the p_1-only executions from $C_0\alpha_h\gamma_{V'_1}$ and $C_0\alpha_h\gamma_{V'_2}$ in which p_1 finishes its ongoing *Scan* operation (if it has one) and then performs a complete *Scan*. The complete *Scan* operation in p_1's solo execution from $C_0\alpha_h\gamma_{V'_1}$ returns the vector V'_1 and the complete *Scan* operation in p_1's solo execution from $C_0\alpha_h\gamma_{V'_2}$ returns the vector $V'_2 \neq V'_1$. Since p_1 takes no steps in $\gamma_{V'_1}$ or $\gamma_{V'_2}$, it must be true that $C_0\alpha_h\gamma_{V'_1} \overset{p_1}{\not\sim} C_0\alpha_h\gamma_{V'_2}$. Therefore, at least one base object must have different values in $C_0\alpha_h\gamma_{V'_1}$ and $C_0\alpha_h\gamma_{V'_2}$.

Lemma 6 (d) implies that, for every p_0-only, $\{V_1, \ldots, V_h\}$-absent execution γ from C_0 and every $B_x \in \mathcal{B}$, we have $value(B_x, C_0\alpha_h\gamma) \notin \mathcal{X}_h(B_x)$. Thus, there are at most $b_x - |\mathcal{X}_h(B_x)|$ possible values for any base object B_x in $C_0\alpha_h\gamma$. This means that the shared objects can hold $\prod_{x=1}^{|\mathcal{B}|}(b_x - |\mathcal{X}_h(B_x)|)$ distinct sequences of values after p_0-only, $\{V_1, \ldots, V_h\}$-absent executions from $C_0\alpha_h$. Since $\gamma_{V'}$ is a p_0-only, $\{V_1, \ldots, V_h\}$-absent execution for all $V' \in \mathcal{V}'$, we have

$$\prod_{x=1}^{|\mathcal{B}|}\big(b_x - |\mathcal{X}_h(B_x)|\big) \geq |\mathcal{V}'| = \prod_{y=1}^{k} c_y \;\; - h.$$

In Appendix A, we show how this implies that $|\mathcal{B}| \geq \frac{1}{2} \cdot \Big(\sum_{y=1}^{k} \log_{b'} c_y + \frac{h}{b'} - \log_{b'}(h+1)\Big)$. ◀

A specific case that motivated our work in [22] is when $c_1 = \ldots = c_k = b_1 = \ldots = b_{|\mathcal{B}|} = b$. By applying Theorem 7 with $b' = c_1 = \ldots = c_k = b$, we obtain

$$|\mathcal{B}| \geq \frac{1}{2} \cdot (\sum_{y=1}^{k} \log_b b + \frac{h}{b} - \log_b(h+1)).$$

Since $\sum_{y=1}^{k} \log_b b = k$, we have $|\mathcal{B}| \geq \frac{1}{2} \cdot (k + \frac{h}{b} - \log_b(h+1))$. When $n \leq b^k - kb + k$, taking $h = n - 1$ gives us Corollary 8 (a). When $n > b^k - kb + k$, taking $h = b^k - kb + k - 1$ gives us $|\mathcal{B}| \geq \frac{1}{2} \cdot \big(b^{k-1} + \frac{k-1}{b} - \log_b(b^k - bk + k)\big)$. Since $\log_b(b^k - bk + k) \leq \log_b b^k = k$, this gives us Corollary 8 (b).

▶ **Corollary 8.** *If the domain sizes of every component $O[1], \ldots, O[k]$ and the domain sizes of the base objects $B_1, \ldots, B_{|\mathcal{B}|}$ are all equal to b, then*
(a) $|\mathcal{B}| \geq \frac{1}{2} \cdot (k + \frac{n-1}{b} - \log_b n)$ *when* $n \leq b^k - bk + k$*, and*
(b) $|\mathcal{B}| \geq \frac{1}{2} \cdot \big(b^{k-1} - \frac{(b-1)k+1}{b}\big)$ *when* $n > b^k - bk + k$.

6 Conclusion

When the domain sizes of the components and base objects used by the implementation are all equal to b and $n \leq b^k - bk + k$, our obstruction-free, single-updater lower bound of $\frac{1}{2} \cdot (k + \frac{n-1}{b} - \log_b \frac{n-1}{2})$ asymptotically matches our obstruction-free, multi-updater upper bound of $k + \lceil \frac{n}{b-1} \rceil$. For all values of n, we conjecture that $k + \lceil \frac{n}{b} \rceil$ base objects with domain size b are required by any obstruction-free, multi-updater, n-process implementation of a scannable object with k fully reusable components that have domain size b. When $n > b^k - bk + k$, we gave an obstruction-free, single-updater lower bound of $\frac{1}{2} \cdot (b^{k-1} - \frac{(b-1)k+1}{b})$ base objects. If b is a constant, then this asymptotically matches the wait-free, single-updater implementation from b^k binary registers in Section 1. This means that, in order to prove a space lower bound better than b^k for larger values of n, we need to consider more complex executions that contain concurrent *Apply* operations. We may also be able to improve our lower bound by considering stronger progress requirements like lock-freedom or wait-freedom.

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, September 1993. `doi:10.1145/153724.153741`.

2 James H. Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, April 1993. `doi:10.1007/BF02242703`.

3 James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Faith Ellen. Limited-use atomic snapshots with polylogarithmic step complexity. *J. ACM*, 62(1):3:1–3:22, 2015. doi:10.1145/2732263.

4 James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, 1990. doi:10.1016/0196-6774(90)90021-6.

5 James Aspnes and Maurice Herlihy. Wait-free data structures in the asynchronous PRAM model. In *Second Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 340–349, July 1990.

6 Hagit Attiya and Faith Ellen. *Impossibility Results for Distributed Computing*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2014. doi:10.2200/S00551ED1V01Y201311DCT012.

7 Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3):121–132, March 1995. doi:10.1007/BF02242714.

8 Hagit Attiya, Nancy Lynch, and Nir Shavit. Are wait-free algorithms fast? *J. ACM*, 41(4):725–763, July 1994. doi:10.1145/179812.179902.

9 Hagit Attiya and Ophir Rachman. Atomic snapshots in o(n log n) operations. *SIAM J. Comput.*, 27(2):319–340, 1998. doi:10.1137/S0097539795279463.

10 Zohir Bouzid, Michel Raynal, and Pierre Sutra. Anonymous obstruction-free (n, k)-set agreement with $n - k + 1$ atomic read/write registers. *Distrib. Comput.*, 31(2):99–117, April 2018. doi:10.1007/s00446-017-0301-7.

11 J.E. Burns and N.A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, 1993. doi:10.1006/inco.1993.1065.

12 Tian Ze Chen and Yuanhao Wei. Step-optimal implementations of large single-writer registers. *Theoretical Computer Science*, 826-827:40–50, 2020. Special issue on OPODIS 2016. doi:10.1016/j.tcs.2020.04.008.

13 Faith Ellen, Panagiota Fatourou, and Eric Ruppert. Time lower bounds for implementations of multi-writer snapshots. *J. ACM*, 54(6):30–es, December 2007. doi:10.1145/1314690.1314694.

14 Faith Ellen, Rati Gelashvili, Nir Shavit, and Leqi Zhu. A complexity-based classification for multiprocessor synchronization. *Distributed Computing*, 33(2):125–144, April 2020. doi:10.1007/s00446-019-00361-3.

15 Rachid Guerraoui and Eric Ruppert. Anonymous and fault-tolerant shared-memory computing. *Distributed Comput.*, 20(3):165–177, 2007. doi:10.1007/s00446-007-0042-0.

16 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991. doi:10.1145/114005.102808.

17 Jaap-Henk Hoepman and John Tromp. Binary snapshots. In *Proceedings of the 7th International Workshop on Distributed Algorithms*, WDAG '93, pages 18–25, Berlin, Heidelberg, 1993. Springer-Verlag.

18 Michiko Inoue and Wei Chen. Linear-time snapshot using multi-writer multi-reader registers. In Gerard Tel and Paul M. B. Vitányi, editors, *Distributed Algorithms, 8th International Workshop, WDAG '94, Terschelling, The Netherlands, September 29 - October 1, 1994, Proceedings*, volume 857 of *Lecture Notes in Computer Science*, pages 130–140. Springer, 1994. doi:10.1007/BFb0020429.

19 A. Israeli, A. Shaham, and A. Shirazi. Linear-time snapshot implementations in unbalanced systems. *Mathematical systems theory*, 28(5):469–486, September 1995. doi:10.1007/BF01185868.

20 Prasad Jayanti. F-arrays: Implementation and applications. In *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 270–279, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/571825.571875.

21 Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM J. Comput.*, 30(2):438–456, April 2000. doi:10.1137/S0097539797317299.

22 Sean Ovens. The space complexity of scannable binary objects. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 509–519, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3465084.3467916.

23 K. Vidyasankar. Converting lamport's regular register to atomic register. *Inf. Process. Lett.*, 28(6):287–290, August 1988. doi:10.1016/0020-0190(88)90175-5.

24 Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time snapshots with applications to concurrent data structures. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, pages 31–46, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3437801.3441602.

25 Leqi Zhu and Faith Ellen. Atomic snapshots from small registers. In Emmanuelle Anceaume, Christian Cachin, and Maria Gradinariu Potop-Butucaru, editors, *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*, volume 46 of *LIPIcs*, pages 17:1–17:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.OPODIS.2015.17.

A Finishing the proof of Theorem 7

Proof. In the proof sketch of Theorem 7 in Section 5, we showed that

$$\prod_{x=1}^{|\mathcal{B}|} \big(b_x - |\mathcal{X}_h(B_x)|\big) \geq |\mathcal{V}'| = \prod_{y=1}^{k} c_y \; - h. \tag{2}$$

Taking the base b' logarithm of both sides of this inequality gives us the following.

$$\sum_{x=1}^{|\mathcal{B}|} \log_{b'} \big(b_x - |\mathcal{X}_h(B_x)|\big) \geq \log_{b'} \Big(\prod_{y=1}^{k} c_y - h\Big) \tag{3}$$

Let $c = \prod_{y=1}^{k} c_y$. We will now show that $\log_{b'}(c-h) \geq \log_{b'} c - \log_{b'}(h+1)$. Notice that $\log_{b'}(c-h) - (\log_{b'} c - \log_{b'}(h+1)) = \log_{b'} \frac{(h+1)\cdot(c-h)}{c}$. In order to show that $\log_{b'} \frac{(h+1)\cdot(c-h)}{c} \geq 0$, it suffices to show that $(h+1)\cdot(c-h) - c \geq 0$. Notice that $(h+1)\cdot(c-h) - c = hc - h^2 - h = h(c-h-1)$. Since $h \leq \prod_{y=1}^{k} c_y - \sum_{y=1}^{k} c_y + k - 1$, we have $c \geq h+1$. We also have $h \geq 0$. Thus, $h(c-h-1) \geq 0$, which implies that $\log_{b'} \frac{(h+1)\cdot(c-h)}{c} \geq 0$. Therefore, $\log_{b'}(c-h) \geq \log_{b'} c - \log_{b'}(h+1)$. By definition of c, this implies that $\log_{b'}\big(\prod_{y=1}^{k} c_y - h\big) \geq \log_{b'}\big(\prod_{y=1}^{k} c_y\big) - \log_{b'}(h+1)$. Substituting this into (3), we obtain the following.

$$\begin{aligned}\sum_{x=1}^{|\mathcal{B}|} \log_{b'} \big(b_x - |\mathcal{X}_h(B_x)|\big) &\geq \log_{b'} \Big(\prod_{y=1}^{k} c_y\Big) - \log_{b'}(h+1) \\ &= \sum_{y=1}^{k} \log_{b'} c_y - \log_{b'}(h+1).\end{aligned}$$

Dividing by $|\mathcal{B}|$ on both sides of this inequality, we obtain

$$\sum_{x=1}^{|\mathcal{B}|} \frac{1}{|\mathcal{B}|} \cdot \log_{b'} \big(b_x - |\mathcal{X}_h(B_x)|\big) \geq \frac{1}{|\mathcal{B}|} \cdot \Big(\sum_{y=1}^{k} \log_{b'} c_y \; - \log_{b'}(h+1)\Big). \tag{4}$$

Since log is concave, Jensen's inequality implies that

$$\sum_{x=1}^{|\mathcal{B}|} \frac{1}{|\mathcal{B}|} \cdot \log_{b'}\big(b_x - |\mathcal{X}_h(B_x)|\big) \leq \log_{b'}\Big(\sum_{x=1}^{|\mathcal{B}|} \frac{1}{|\mathcal{B}|} \cdot \big(b_x - |\mathcal{X}_h(B_x)|\big)\Big)$$
$$= \log_{b'}\Big(b' - \frac{1}{|\mathcal{B}|} \cdot \sum_{x=1}^{|\mathcal{B}|} |\mathcal{X}_h(B_x)|\Big)$$
$$= \log_{b'}\Big(b' - \frac{h}{|\mathcal{B}|}\Big).$$

Substituting this into (4), we have

$$\log_{b'}\Big(b' - \frac{h}{|\mathcal{B}|}\Big) \geq \frac{1}{|\mathcal{B}|} \cdot \Big(\sum_{y=1}^{k} \log_{b'} c_y \; - \log_{b'}(h+1)\Big).$$

Multiplying by $|\mathcal{B}|$ on both sides of the inequality, we obtain

$$|\mathcal{B}| \cdot \log_{b'}\Big(b' - \frac{h}{|\mathcal{B}|}\Big) \geq \sum_{y=1}^{k} \log_{b'} c_y \; - \log_{b'}(h+1). \tag{5}$$

Let $x = -\big(\frac{h}{|\mathcal{B}|}\big)$. So $\log_{b'}\big(b' - \frac{h}{|\mathcal{B}|}\big) = \log_{b'}(b' + x)$. Since $|\mathcal{B}| \geq \frac{h}{b'-1}$, we have $1 - b' \leq x < 0$. The Maclaurin series expansion of $\log_{b'}(b' + x)$ is the following.

$$1 + \frac{x}{b' \cdot \ln(b')} - \frac{x^2}{2(b')^2 \cdot \ln(b')} + \frac{x^3}{3(b')^3 \cdot \ln(b')} - \frac{x^4}{4(b')^4 \cdot \ln(b')} \cdots$$

The series converges provided $|x| < |b'|$. Since $x < 0$, every term of $-\frac{x^2}{2(b')^2 \cdot \ln(b')} + \frac{x^3}{3(b')^3 \cdot \ln(b')} - \frac{x^4}{4(b')^4 \cdot \ln(b)} \ldots$ is negative. Hence, we have

$$1 + \frac{x}{b' \cdot \ln(b')} \geq \log_{b'}(b' + x). \tag{6}$$

Notice that $\frac{x}{b' \cdot \ln(b')} - \frac{x}{b'} = \frac{x \cdot (1 - \ln(b'))}{b' \cdot \ln(b')} \leq 1$, since $|x| \leq b' - 1$ and $b' \geq 2$. Hence, we have $1 + \frac{x}{b'} \geq \frac{x}{b' \cdot \ln(b')}$. Combined with (6) and the definition of x, this gives us

$$1 + \Big(1 - \frac{h}{|\mathcal{B}| \cdot b'}\Big) \geq 1 - \frac{h}{|\mathcal{B}| \cdot b' \cdot \ln(b')} \geq \log_{b'}\Big(b' - \frac{h}{|\mathcal{B}|}\Big).$$

Combined with (5), this gives us

$$|\mathcal{B}| \cdot \Big(2 - \frac{h}{|\mathcal{B}| \cdot b'}\Big) \geq |\mathcal{B}| \cdot \log_{b'}\Big(b' - \frac{h}{|\mathcal{B}|}\Big) \geq \sum_{y=1}^{k} \log_{b'} c_y \; - \log_{b'}(h+1).$$

Thus, we have

$$2 \cdot |\mathcal{B}| - \frac{h}{b'} \geq \sum_{y=1}^{k} \log_{b'} c_y \; - \log_{b'}(h+1).$$

Add $\frac{h}{b'}$ to both sides and then divide by 2 to obtain

$$|\mathcal{B}| \geq \frac{1}{2} \cdot \Big(\sum_{y=1}^{k} \log_{b'} c_y \; + \frac{h}{b'} - \log_{b'}(h+1)\Big).$$

This concludes the proof of the theorem. ◀

Near-Optimal Distributed Computation of Small Vertex Cuts

Merav Parter ✉
Weizmann Institute, Rehovot, Israel

Asaf Petruschka ✉
Weizmann Institute, Rehovot, Israel

Abstract

We present near-optimal algorithms for detecting small vertex cuts in the CONGEST model of distributed computing. Despite extensive research in this area, our understanding of the *vertex* connectivity of a graph is still incomplete, especially in the distributed setting. To this date, all distributed algorithms for detecting cut vertices suffer from an inherent dependency in the maximum degree of the graph, Δ. Hence, in particular, there is no truly sub-linear time algorithm for this problem, not even for detecting a *single* cut vertex. We take a new algorithmic approach for vertex connectivity which allows us to bypass the existing Δ barrier.

- As a warm-up to our approach, we show a simple $\widetilde{O}(D)$-round[1] randomized algorithm for computing all cut vertices in a D-diameter n-vertex graph. This improves upon the $O(D + \Delta/\log n)$-round algorithm of [Pritchard and Thurimella, ICALP 2008].
- Our key technical contribution is an $\widetilde{O}(D)$-round randomized algorithm for computing all cut *pairs* in the graph, improving upon the state-of-the-art $O(\Delta \cdot D)^4$-round algorithm by [Parter, DISC '19]. Note that even for the considerably simpler setting of *edge* cuts, currently $\widetilde{O}(D)$-round algorithms are currently known *only* for detecting pairs of cut edges.

Our approach is based on employing the well-known linear graph sketching technique [Ahn, Guha and McGregor, SODA 2012] along with the heavy-light tree decomposition of [Sleator and Tarjan, STOC 1981] . Combining this with a careful characterization of the survivable subgraphs, allows us to determine the connectivity of $G \setminus \{x, y\}$ for every pair $x, y \in V$, using $\widetilde{O}(D)$-rounds. We believe that the tools provided in this paper are useful for omitting the Δ-dependency even for larger cut values.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Graph algorithms analysis

Keywords and phrases Vertex-connectivity, Congest, Graph Sketches

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.31

Related Version *Full Version*: `https://www.weizmann.ac.il/math/parter/sites/math.parter/files/uploads/DistributedVertexCutsDISC22.pdf`

Funding *Merav Parter*: Supported by the European Research Council (ERC) No. 949083), and by the Israeli Science Foundation (ISF) No. 2084/18.

1 Introduction and Our Contribution

The vertex connectivity of the graph is a central concept in graph theory and extensive attention has been paid to developing algorithms to compute it in various computational models. Recent years have witnessed an enormous progress in our understanding of vertex cuts, from a pure graph theoretic perspective [36] to many algorithmic applications [30, 28, 36, 20].

[1] Throughout the paper, we use the notation $\widetilde{O}$ to hide poly-logarithmic in n terms.

© Merav Parter and Asaf Petruschka;
licensed under Creative Commons License CC-BY 4.0

Despite this exciting movement, our algorithmic toolkit for handling vertex cuts is still somewhat limited. A large volume of the work, in the centralized setting, has focused on fast algorithms for detecting minimum vertex cuts of size at most k, for some small number k. Until recently, near-linear time algorithms where known only for $k \leq 2$ [39, 21]. A sequence of recent breakthrough results [5, 28] provides an almost-linear time sequential algorithm for computing the vertex connectivity (even for large connectivity values).

As we see soon, the situation is considerably worse in distributed settings, where the problem is still fairly open already for $k = 1$. Throughout, we consider the CONGEST model [35]. In this model, each node holds a processor with a unique and arbitrary ID of $O(\log n)$ bits, and initially only knows the IDs of its neighbors in the graph. The execution proceeds in synchronous rounds and in each round, each node can send a message of size $O(\log n)$ to each of its neighbors. The primary complexity measure is the number of communication rounds. For n-vertex D-diameter graphs, Pritchard and Thurimella [37] presented a randomized algorithm for detecting a (single) cut vertex (a.k.a articulation point) within $O(D + \Delta/\log n)$ CONGEST rounds, where Δ is the maximum degree of the graph. [37] conclude their paper by noting:

> [37] *It would be interesting to know if our distributed cut vertex algorithm could be synthesized with the cut vertex algorithm of [40] to yield further improvement. Alternatively, a lower bound showing that no $O(D)$-time algorithm is possible for finding cut vertices would be very interesting.*

No progress on the complexity of this problem has been done since then. For small cut values k, Parter [33] employed the well-known fault-tolerant sampling technique [42, 25] for detecting k vertex cuts in $(\Delta \cdot D)^{\Theta(k)}$ deterministic rounds. Turning to approximation algorithms, for $k = \Omega(\log n)$, Censor-Hillel, Ghaffari and Kuhn [4] provided a $O(\log n)$ approximation for computing the *value* of the vertex connectivity of the graph within $\widetilde{O}(D + \sqrt{n})$ rounds. [4] also presented a lower bound of $\widetilde{\Omega}(D + \sqrt{n/k})$ V-CONGEST rounds. In the V-CONGEST model, each *node* (rather than an edge) is restricted to send only $O(\log n)$ bits, in total, in every round. As shown in this paper, this lower bound does not hold in the standard CONGEST model.

We follow the terminology of [37]: a *cut vertex* is a vertex x such that $G \setminus \{x\}$ is not connected. A *cut pair* is a pair of vertices x, y such that $G \setminus \{x, y\}$ is not connected. For brevity we call these objects, small cuts. Our main results in this paper are near-optimal algorithms for detecting these small cuts, in the sense that for every small cut, there is at least one vertex in the graph that learns it. Our first contribution is in presenting a (perhaps surprisingly) simple randomized algorithm[2] that can detect all cut vertices in the graph in $\widetilde{O}(D)$ rounds. The edge-congestion[3] of the algorithm is $\widetilde{O}(1)$ bits[4].

▶ **Theorem 1.** *There is a randomized algorithm that w.h.p. identifies all single cut-vertices in G within $\widetilde{O}(D)$ rounds. The edge congestion is $\widetilde{O}(1)$. In the output, each vertex $x \in V$ learns if it is a cut vertex.*

---

[2] As usual, all presented randomized algorithms in this paper have success guarantee of $1 - 1/n^c$, for any given constant $c > 1$.

[3] The edge congestion of a given algorithm is the worst-case bound on the total number of messages exchanged through a given edge e in the graph.

[4] We exploit this bounded congestion for detecting cut pairs.

This settles the question raised by [37]. Our algorithm is based on the well-known *graph-sketching* technique of Ahn et al. [1]. This technique has admitted numerous applications in the context of connectivity computation under various computational settings, e.g., [23, 22, 18, 26, 29, 17, 11, 8]. Yet, to the best of our knowledge, it has not been employed before in the context of CONGEST algorithms for minimum vertex-cut computation.

We then turn to consider the problem of detecting cut pairs. It has been noted widely in the literature that there is a sharp qualitative difference between a single failure and two failures. This one-to-two jump has been accomplished by now for a wide-variety of fault-tolerant settings, e.g., reachability oracles [6], distance oracles [9], distance preservers [32, 19, 34] and vertex-cuts [21, 2, 3, 13]. While it is relatively easy to extend our algorithm of Theorem 1 to detect cut pairs in $\widetilde{O}(D^2)$ rounds, providing a near-optimal complexity of $\widetilde{O}(D)$ rounds, turns out to be quite involved. Our key technical contribution is:

▶ **Theorem 2.** *There is a randomized algorithm that w.h.p. identifies all cut pairs in G within $\widetilde{O}(D)$ rounds. For each cut pair x, y, either x or y learns that fact.*

We observe that even for the simpler problem of edge-connectivity (see Remark below), an $\widetilde{O}(D)$-round algorithm is currently only known for edge cuts of size at most *two* due to [37]. Hence, we are now able to match the complexity of these two problems for small cut values. Our algorithm is based on distinguishing between two structural cases depending on the locations of the cut pair x, y in a BFS tree of G. The first case which we call *dependent* handles the setting where the x and y have ancestry/descendant relations. The second *independent* case assumes that x and y are not dependent, i.e., $\mathrm{LCA}(x, y) \notin \{x, y\}$, where $\mathrm{LCA}(x, y)$ is the lowest/least common ancestor of x and y in the BFS tree. Each of these cases call for a different approach. We believe that the tools provided in this paper should hopefully pave the way towards detecting larger vertex-cuts with no dependency in the maximum degree Δ (as it is the case for the state-of-the-art algorithm by [33]). For a more in-depth technical overview, see Sec. 1.1.

Remark on the Edge-Connectivity. It is widely known that in undirected graphs, vertex connectivity and vertex cuts are significantly more complex than edge connectivity and edge cuts, for which now the following result are known: an $\widetilde{O}(m)$-time centralized exact algorithm [24, 16, 12] and an $\widetilde{O}(D + \sqrt{n})$ exact distributed algorithms [7]. For constant values of edge-connectivity a $\mathrm{poly}(D)$-round algorithm is given in [33].

1.1 Our Approach, in a Nutshell

We provide the key ideas of our algorithms. Our end goal is to simulate a connectivity algorithm in the graph $G \setminus \{x, y\}$ for *every* pair of vertices $x, y \in V$. Note that this is not trivial already for a single x, y pair as the diameter of the subgraph $G \setminus \{x, y\}$ might be as large as $\Omega(\Delta D)$, hence using on-shelf connectivity algorithms lead to a round complexity of $O(\min\{D + \sqrt{n}, \Delta D\})$. We bypass this Δ dependency by using the edges incident to the vertices x, y as *shortcuts*. Then, to minimize the congestion imposed by running possibly n^2 connectivity algorithms in parallel, we employ a preprocessing phase in which we collect *graph-sketch* information (explained next) at each vertex x. This information allows each vertex x to pinpoint at a bounded number of cut-mate suspects. In addition, it allows x, in certain cases, to locally simulate connectivity queries without using further communication. Throughout, let T be a BFS tree rooted at some source s, and denote the T-paths by $\pi(\cdot, \cdot)$.

We start by employing the well-known *heavy-light tree decomposition* technique by Sleator and Tarjan [38]. This classifies the edges of T into light and heavy edges. The useful properties are that each vertex v has $O(\log n)$ light edges on its tree path $\pi(s, v)$, and in addition, each v is the parent of *one* heavy edge, connecting v to its unique heavy child. It is easy to compute this decomposition on T in $\widetilde{O}(D)$ rounds. For a vertex x, let T_x be the subtree of T rooted at x.

Basic Tools: Graph Sketches and Borůvka Algorithm. A *graph sketch* of a vertex v is a randomized string of $\widetilde{O}(1)$ bits that compresses v's edges [1]. The linearity of these sketches allows one to infer, given the sketches of subset of vertices S, an outgoing cut edge $(S, V \setminus S)$ with constant probability. A common approach for deducing the graph connectivity merely from the sketches of the vertices is based on the well-known Borůvka algorithm [31]. This algorithm works in $O(\log n)$ phases, where in each phase, from each growable component an outgoing edge is selected. All these outgoing edges are added to the forest, while ignoring cycles. Each such phase reduces the number of growable components by constant factor, thus within $O(\log n)$ phases a maximal forest is computed. Since this algorithm only requires the computation of outgoing edges it can simulated using $O(\log n)$ independent sketches for each of the vertices. In our algorithms, we aggregate graph sketches over the BFS tree T which allows the vertices x to locally simulate Borůvka in the graph $G \setminus \{x\}$. This is illustrated in our algorithm for detecting a single cut vertex, described next.

Warm Up: Detecting Single Cut Vertices. Our algorithm starts by letting each vertex v locally compute its individual $\mathsf{Sketch}_G(v)$. Then, by aggregating the sketches (using their linearity) from the leaf vertices to the root s over the BFS tree T, each vertex v learns its subtree-sketch $\mathsf{Sketch}_G(V(T_v))$. Once this is completed, it is easy to let each vertex $x \in V$ learn the G-sketch information of all the connected components in $T \setminus \{x\}$. We then show that x can locally modify these G-sketches into $(G \setminus \{x\})$-sketches. At this point, the vertex x can locally apply the Borůvka algorithm in $G \setminus \{x\}$ and deduce if $G \setminus \{x\}$ is connected. The full details (proof of Theorem 1) appear in Appendix A.

We now turn to consider the considerably more challenging task of detecting cut-pairs. We classify these pairs into dependent and independent.

Detecting Dependent Cut Pairs. Our approach for the dependent case is based on designing algorithms $\{\mathcal{A}_y\}_{y \in V}$, where $\mathcal{A}_y$ detects all xy cut pairs of the form $x \in T_y$. We show that each such an algorithm $\mathcal{A}_y$ can be designed in a way that sends a total of $\widetilde{O}(1)$ messages only along edges incident to $V(T_y)$, and runs in $\widetilde{O}(D)$ rounds. The standard random delay technique allows us then to schedule the execution of all n algorithms $\{\mathcal{A}_y\}_{y \in V}$ within $\widetilde{O}(D)$ rounds. At a high level, each algorithm $\mathcal{A}_y$ is based on employing the single-vertex cut algorithm in the graph $G \setminus \{y\}$. Our challenge is then twofold: first, the diameter of the graph $G \setminus \{y\}$ might be as large as $\Omega(\Delta D)$, and second, communication is restricted to use only edges incident to $V(T_y)$. We overcome these challenges by using y as a coordinator, providing global computation services and communication shortcuts that essentially enables efficient simulation (in both dilation and congestion) of the vertex cut algorithm in $G \setminus \{y\}$.

Detecting Independent Cut Pairs. The most technically involved case is where x, y are independent, namely, *do not* have ancestry relations in T. A-priori, the number of such potential cut-mates y for a given vertex x might be even linear in n. To filter out irrelevant options, the algorithm starts by computing at each vertex x a tree $\widehat{T}_x$ that encodes the

connectivity between s and the vertices in $V_x = V(T_x) \setminus \{x\}$ in the graph $G \setminus \{x\}$. Let $\mathcal{C}_x = \{C_1, \ldots, C_k\}$ denote the collection of maximal connected components in the graph $G[V_x]$. The tree $\widehat{T}_x$ consists of k paths of the form $\pi(s, u_C) \circ (u_C, v_C)$ for every component $C \in \mathcal{C}_x$, where v_C is some representative vertex in C. It is then easy to observe that the potential cut mates y must appear on the paths $\{\pi(s, u_C) \mid C \in \mathcal{C}_x\}$. For a given suspect y, we call the $\mathcal{C}_x$-components C for which $y \in \pi(s, u_C)$, *y-sensitive*. Our argument has the following structure.

Multiple xy-Connectivity Algorithms, Under a Promise. For a fixed xy pair, we design an algorithm $A^P_{x,y}$ that determines the connectivity in $G \setminus \{x, y\}$ *given* an x-y path $\Pi_{x,y}$ (on which x, y can exchange messages). The algorithm $A^P_{x,y}$ has the special property that it sends messages either along $\Pi_{x,y}$, or else along edges incident to a restricted subset of vertices in T_x, T_y, defined as follows. Let $\mathsf{LDS}(x, y) \subset V(T_x)$ be the set of all vertices which are descendants of the *light* children of x, and belong to a y-sensitive component in $\mathcal{C}_x$. The set $\mathsf{LDS}(y, x)$ is defined in an analogous manner. The algorithm $A^P_{x,y}$ is then guaranteed to send $\widetilde{O}(1)$ messages only along $\Pi_{x,y}$ and along edges incident to the vertices of $\mathsf{LDS}(x, y) \cup \mathsf{LDS}(y, x)$. This restriction is crucial in order to run multiple $A^P_{x,y}$ algorithms, for distinct x, y, in parallel. Using the properties of the heavy-light tree decomposition and our sensitivity definition, one can show that each vertex $w \in V$ belongs to the $\mathsf{LDS}(x, y)$ sets of at most $\widetilde{O}(D)$ pairs xy. The main challenge is in bounding the overlap between the $\Pi_{x,y}$ paths, cross distinct xy pairs. We show that given a subset $Q \in V \times V$, the collection of $\{A^P_{x,y} \mid (x, y) \in Q\}$ algorithms can be scheduled in parallel in $\widetilde{O}(D)$ rounds, given that following promise holds for Q:

> [**Promise**:] *There is a path collection $\mathcal{P}_Q = \{\Pi_{x,y} \mid (x, y) \in Q\}$ such that each path has length $O(D)$, and each edge appears on $\widetilde{O}(D)$ paths in $\mathcal{P}_Q$.*

One can show, using the properties of heavy-light decomposition, that each vertex belongs to the $\mathsf{LDS}(x, y)$ sets of at most $\widetilde{O}(D)$ pairs x, y. Hence, by combining this fact with the promise, the algorithms for all the Q pairs can be run in parallel, using the random delay approach [27, 14].

On a high level, each algorithm $A^P_{x,y}$ works by letting x and y jointly simulating the Borůvka algorithm in $G \setminus \{x, y\}$. The main challenge is that the communication is restricted to the edges incident to $\mathsf{LDS}(x, y) \cup \mathsf{LDS}(y, x)$, despite the fact that one should also take into account the remaining vertices in T_x, T_y, e.g., descendants of the *heavy* children of x, y. In each Borůvka phase, we maintain the invariant that x, y jointly hold the sketches of connected-subsets (denoted as *parts*) in $G \setminus \{x, y\}$, where we split the responsibility between x, y in a careful manner. We mainly distinguish between parts that contain a heavy child of x, y and the remaining *light* parts that are contained in $\mathsf{LDS}(x, y) \cup \mathsf{LDS}(y, x)$. The merges of the light parts are implemented by using communication between vertices in $\mathsf{LDS}(x, y) \cup \mathsf{LDS}(y, x)$. The merges concerning the heavy parts are implemented by using the direct xy communication over the $\Pi_{x,y}$ path. Each such Borůvka phase is implemented in $\widetilde{O}(D)$ rounds. At the end of the simulation, x, y both learn whether $G \setminus \{x, y\}$ is connected.

Omitting the Promise Based on Classification Into Light and Heavy Independent Pairs. While the promise clearly holds for $\widetilde{O}(D)$ pairs, it clearly does not hold for all n^2 pairs, in general. Our approach is based on classifying the collection of the xy pairs into two classes: *light* and *heavy*. This classification is based on the trees $\widehat{T}_x, \widehat{T}_y$, as well as on the heavy-light decomposition of T. Informally, for a light pair xy, one can define a $\Pi_{x,y}$ that intersects

a light subtree of either x or y. These paths can be shown to have a bounded overlap, hence satisfying the promise. Handling the heavy pairs is more involved. Here we take a mixed approach. We define a special subset of the heavy pairs for which the promise can be satisfied (denoted as *mutual pairs*). This subset is chosen in a careful way that guarantees the following, perhaps surprising, property: the remaining (not mutual) heavy pairs x, y can be decided locally, at either x or y. Our key observation is that for a xy heavy pair, the graph $G \setminus \{x, y\}$ is connected iff one of the heavy children of x, y is connected to s in $G \setminus \{x, y\}$. Hence, it is mainly essential for x, y to collect a sketch information on the components of these heavy children in $\mathcal{C}_x, \mathcal{C}_y$. This information can be then aggregated over T. The formal implementation of this step (completing the proof of Theorem 2) appears in the full version of the paper.

1.2 Preliminaries

Throughout the paper, we fix a connected n-vertex graph $G = (V, E)$, and a BFS tree T for G rooted at some arbitrary source vertex $s \in V$. We denote the unique tree path from u to v by $\pi(u, v, T)$. When the tree T is clear from context, we may omit it and simply write $\pi(u, v)$. We use the $\circ$ operator for path-concatenation. An (undirected) edge between vertices u,v is denoted by (u, v). For $x, y \in V$, a vertex subset $S \subseteq V$ is said to be xy-connected if all the vertices of S belong to the same connected component of $G \setminus \{x, y\}$.

Heavy-Light Tree Decomposition. We now present our heavy-light terminology, the notion of *compressed paths*, and their distributed computation.

▶ **Definition 3** (Heavy-light decomposition). *For a non-leaf vertex $v \in V(T)$, its* heavy child*, denoted v_h, is the child v' of v maximizing*[5] *the number of vertices in its subtree $T_{v'}$. Any other v-child of v is a* light child*. A tree vertex is* heavy *if it is the heavy child of its parent, and* light *otherwise (so the root s is light). A tree edge is* heavy *if it connects a vertex to its heavy child, and* light *otherwise. If (u, u') is a heavy (resp., light) edge in the path $\pi(s, v)$, then u is a* heavy ancestor *(resp.,* light ancestor*) of v, and v is a* heavy descendant *(resp,* light descendant*) of u. (Note that e.g. a "heavy ancestor" need not be a heavy vertex itself.) We denote by $\mathsf{LA}(v)$ (resp., $\mathsf{LD}(v)$) the set of v's light ancestors (resp., descendants). It is easy to show that $\pi(s, v, T)$ contains $O(\log n)$* light *vertices/edges, hence also $|\mathsf{LA}(v)| = O(\log n)$.*

▶ **Definition 4** (Compressed paths). *Let $v \in V(T)$. Let $L = [s = v_0, v_1, \ldots, v_k]$ be the ordered list of the light vertices on the root-to-v path $\pi(s, v, T)$. The* compressed path *of v with respect to T, denoted $\pi^*(s, v, T)$ consists of the list L, along with a table mapping each v_i to the number of heavy vertices appearing between v_i and v_{i+1} in $\pi(s, v, T)$ (where we define $v_{k+1} = v$). Note that the compressed path $\pi^*(s, v, T)$ has bit-length $O(\log^2 n)$.*

Observe that the compressed paths can be used as ancestry labels *in T: Given the compressed path $\pi^*(s, u, T)$ and $\pi^*(s, v, T)$, one can check whether $\pi(s, u, T)$ is a prefix of $\pi(s, v, T)$, and hence determine whether u is an ancestor of v.*

In our context of distributed computation, we have the following lemma. Missing proofs in this section appear in the full version of the paper.

▶ **Lemma 5.** *For every tree T, there is an $\widetilde{O}(D(T))$-rounds $\widetilde{O}(1)$-congestion algorithm letting each vertex v of T learn its heavy/light classification and its compressed path $\pi^*(s, v, T)$.*

[5] Ties are broken arbitrarily and consistently.

Graph Sketches. We now give a formal but brief definition of graph sketches. We follow [8], and refer the reader to Section 3.2.1 therein for a detailed presentation of the subject. Throughout, let $\oplus$ denote the bitwise-XOR operator. The first required ingredients are randomized *unique edge identifiers*:

▶ **Lemma 6** (Modification of Lemma 3.8 in [8]). *Using a random seed $\mathcal{S}_{ID}$ of $O(\log^2 n)$ random bits, one can compute a collection of $M = \binom{n}{2}$ $O(\log n)$-bit identifiers for the pairs in $\binom{V}{2}$, denoted $\mathcal{I} = \{\mathrm{UID}(e_1), \ldots, \mathrm{UID}(e_M)\}$, with the following property: For any nonempty subset $E' \subseteq E$, $\Pr[\oplus_{e \in E'} \mathrm{UID}(e) \in \mathcal{I}] \leq 1/n^{10}$. Furthermore, for any $e = (u, v)$, the identifier $\mathrm{UID}(e)$ can be computed from $\mathrm{ID}(u)$, $\mathrm{ID}(v)$ and the random seed $\mathcal{S}_{ID}$.*

Next, we define the notion of *extended edge identifiers*, formed by augmenting the $\mathrm{UID}(e)$ with the IDs and the T-ancestry labels of the endpoints based on compressed paths, namely $\mathsf{ANC}_T(v) = \pi^*(s, v, T)$. Formally, an edge $e = (u, v)$ we have

$$\mathrm{EID}_T(e) = [\mathrm{UID}(e), \mathrm{ID}(u), \mathrm{ID}(v), \mathsf{ANC}_T(u), \mathsf{ANC}_T(v)] \ . \tag{1}$$

Equipped with these definitions, we are ready to define the sketches. We now follow [10, 11, 8] and use pairwise independent hash functions for this purpose. Choose $L = c \log n$ pairwise independent hash functions $h_1, \ldots, h_L : \{0,1\}^{\Theta(\log n)} \to \{0, \ldots, 2^{\log M} - 1\}$, and for each $i \in \{1, \ldots, L\}$ and $j \in [0, \log M]$ define the edge set $E_{i,j} = \{e \in E \mid h_i(e) \in [0, 2^{\log M - j})\}$. Each of these hash functions can be defined using a random seed of logarithmic length [41]. Thus, a random seed $\mathcal{S}_h$ of length $O(L \log n)$ can be used to determine the collection of all these L functions. For each vertex v and indices i, j, let $E_{i,j}(v)$ be the edges incident to v in $E_{i,j}$. The ith *basic sketch unit* of each vertex v is then given by:

$$\mathsf{Sketch}_{G,i}(v) = [\oplus_{e \in E_{i,0}(v)} \mathrm{EID}_T(e), \ldots, \oplus_{e \in E_{i,\log M}(v)} \mathrm{EID}_T(e)].$$

We extend the sketches to be defined on vertex subsets by XORing. Namely, for every subset of vertices S, we define $\mathsf{Sketch}_{G,i}(S) = \oplus_{v \in S} \mathsf{Sketch}_{G,i}(v)$. The *sketch* of each vertex v is defined by a concatenation of $L = \Theta(\log n)$ basic sketch units:

$$\mathsf{Sketch}_G(v) = [\mathsf{Sketch}_{G,1}(v), \mathsf{Sketch}_{G,2}(v), \ldots \mathsf{Sketch}_{G,L}(v)] \ .$$

Again, we extend this definition to vertex subsets $S \subseteq V$ by $\mathsf{Sketch}_G(S) = \oplus_{v \in S} \mathsf{Sketch}_G(v)$. The main use of graph sketches is in finding outgoing edges:

▶ **Lemma 7** (Modification of Lemma 3.11 in [8]). *For any subset S, given a basic sketch unit $\mathsf{Sketch}_{G,i}(S)$ and the seed $\mathcal{S}_{ID}$ one can compute, with constant probability*[6] *$\mathrm{EID}_T(e)$ for an outgoing edge e from S in G, if such exists.*

▶ **Lemma 8.** *Let $S \subseteq V$, and let $E' \subseteq E$ be a set of outgoing edges from S. Then, given $\mathsf{Sketch}_G(S)$, the random seed $\mathcal{S}_h$, and the extended identifiers $\mathrm{EID}_T(e)$ of all $e \in E'$, one can compute the $\mathsf{Sketch}_{G \setminus E'}(S)$.*

Distributed Scheduling. The congestion of an algorithm $\mathcal{A}$ is defined by the worst-case upper bound on the number of messages exchanged through a given graph edge when simulating $\mathcal{A}$. Throughout, we make an extensive use of the following random delay approach of [27], adapted to the CONGEST model.

[6] Over the choice of the random seeds $\mathcal{S}_{ID}$ and $\mathcal{S}_h$.

▶ **Theorem 9** ([14, Theorem 1.3]). *Let G be a graph and let $\mathcal{A}_1, \ldots, \mathcal{A}_m$ be m distributed algorithms, each algorithm takes at most* $\mathtt{d}$ *rounds, and where for each edge of G, at most* $\mathtt{c}$ *messages need to go through it, in total* over all *these algorithms. Then, there is a randomized distributed algorithm that w.h.p. runs all the algorithms in $\widetilde{O}(\mathtt{c}+\mathtt{d})$ rounds.*

2 Dependent Cut Pairs

In this section we present an $\widetilde{O}(D)$-rounds distributed algorithm for detecting *dependent* cut pairs in G, i.e. pairs xy where x is a descendant of y in the BFS tree T rooted at s. Recall that our approach is based on scheduling the execution of algorithms $\{\mathcal{A}_y\}_{y\in V}$, where $\mathcal{A}_y$ detects all cut pairs xy such that $x \in T_y$ (see Section 1.1). By employing the single-vertex cut algorithm from Section A as a common preprocessing phase prior to the execution of the $\{\mathcal{A}_y\}_{y\in V}$ algorithms, we may assume that there are no 1-vertex cuts in G. Furthermore, by carefully examining the properties of this algorithm, we may assume that every $v \in V$ holds the following preprocessing information:

- The random seeds $\mathcal{S}_{ID}$ and $\mathcal{S}_h$.
- $\mathrm{EID}_T(e)$ for every edge e incident to v.
- $|V(T_v)|$ and $|V(T_{v'})|$ for every T-child v' of v.
- $\mathsf{Sketch}_G(v)$, $\mathsf{Sketch}_G(V)$, $\mathsf{Sketch}_G(V(T_v))$ and $\mathsf{Sketch}_G(V(T_{v_i}))$ for every T-child v' of v.
- An edge set $\widetilde{E}(v) \subseteq E \setminus E(T)$ such that $\widetilde{T}(v) = (T \setminus \{v\}) \cup \widetilde{E}(v)$ is a spanning tree of $G \setminus \{v\}$. For each $e \in \widetilde{E}(v)$, its extended identifier $\mathrm{EID}_T(e)$ is known.

We next describe the algorithms $\mathcal{A}_y$:

▶ **Lemma 10.** *Assuming all vertices know their preprocessing information, there is an $\widetilde{O}(D)$-rounds $\widetilde{O}(1)$-congestion algorithm $\mathcal{A}_y$ that detects all cut pairs xy where $x \in V(T_y)$. The algorithm $\mathcal{A}_y$ sends messages only on edges incident to $V(T_y)$.*

Step 0: Local Computation of Component Tree for $\widetilde{T}(y)$ in y. Throughout, let $\widetilde{E} = \widetilde{E}(y)$ and $\widetilde{T} = \widetilde{T}(y)$, and denote the T-children of y by $y_1, \ldots, y_k$. This preliminary step is executed by local computation in y. It constructs the *component tree* $\widetilde{CT}$ in which every connected component of $\widetilde{T} \setminus \widetilde{E}$ is contracted into a single node. Note that $\widetilde{T} \setminus \widetilde{E} = T \setminus \{y\}$, namely the nodes in $\widetilde{CT}$ correspond to connected components of $T \setminus \{y\}$. More concretely, for every $i = 1, \ldots, k$ the component $C_i = V(T_{y_i})$ is a node of $\widetilde{CT}$, and (unless $y = s$) there is another node for the component $C_0 = V(T) \setminus V(T_y)$. Each edge (C_i, C_j) in $\widetilde{CT}$ correspond to the unique $\widetilde{E}$-edge incident to both C_i and C_j. Observe that the extended edge identifiers known to y by preprocessing contain the T-ancestry labels of all endpoints of $\widetilde{E}$-edges, as well as those of the y_i's. Using these ancestry labels, y can determine the components incident to each edge $e \in \widetilde{E}$, and therefore construct $\widetilde{CT}$.

For clarity of presentation we assume $y \neq s$; the special case $y = s$ is easier, and requires only slight modifications. We set s as the root of $\widetilde{T}$, and accordingly C_0 is the root of $\widetilde{CT}$. For each $i = 1, \ldots, k$, denote by $e_i = (r_i, p_i)$ the unique edge in $\widetilde{E}$ connecting C_i to its parent in $\widetilde{CT}$, where r_i is the endpoint of e_i inside C_i, and p_i is the endpoint lying in the parent component. See Fig. 1 in Appendix B for an illustration.

Step 1: Construction of $\widetilde{T}$. The goal of this step is for each vertex in $V(T_y) \setminus \{y\}$ to learn its parent in $\widetilde{T}$. First, y sends its children their corresponding edges from $\widetilde{E}$, namely each y_i learns $\mathrm{EID}_T(e_i)$. The y_i's then propagate (in parallel) their received edges down their T-subtrees, so that for all $i = 1, \ldots, k$, all the vertices of component C_i know $\mathrm{EID}_T(y_i)$.

Then, a BFS procedure initilized in r_i is executed inside each tree T_{y_i} (in parallel). This completes the step, since the $\widetilde{T}$-parent of each vertex in C_i is its BFS-parent from this last procedure, except for r_i whose $\widetilde{T}$-parent is p_i.

Step 2: Computing $\widetilde{T}$-Ancestry labels. In later steps, we will locally simulate Borůvka's algorithm similarly to Section A, but with the initial components being parts of $\widetilde{T}$. In order to identify which components get merged by the outgoing edges, we will need ancestry labels with respect to the tree $\widetilde{T}$ rather than T. As we are restricted to send messages only on $V(T_y)$-incident edges, we would like the T- and $\widetilde{T}$-labels to coincide for vertices in C_0 (as some of them cannot be informed of new labels). Note that the compressed paths of $v \in C_0$ w.r.t. T and $\widetilde{T}$ are generally different, even though $\pi(s, v, T) = \pi(s, v, \widetilde{T})$, as the these trees have different heavy-light notions. Hence, instead of relying solely on compressed paths in $\widetilde{T}$, we take a hybrid approach and define new labels based on breaking each $\widetilde{T}$-path to a T-part and a strictly $\widetilde{T}$-part, and compressing them accordingly. We still have the challenge of computing (at least part of) the heavy-light decomposition of $\widetilde{T}$. As the diameter of $\widetilde{T}$ might be $\Omega(\Delta D)$, we cannot use simple bottom-up or top-down computations on $\widetilde{T}$. The key for overcoming this is utilizing y as a coordinator, enabling the parts C_i to work in parallel. This yields the following claim. Missing proofs in this section appear in the full version of the paper.

▷ Claim 11. In $\widetilde{O}(D)$-rounds of computation with $\widetilde{O}(1)$ congestion, in which messages are sent only on $V(T_y)$-incident edges, one can compute $\widetilde{T}$-ancestry labels $\mathsf{ANC}_{\widetilde{T}}(\cdot)$ of $\widetilde{O}(1)$ bits, such that *every* vertex v of $\widetilde{T}$ learns $\mathsf{ANC}_{\widetilde{T}}(v)$.

Step 3: Computing Sketches w.r.t. $G \setminus \{y\}$ and $\widetilde{T}$. First, we define new extended edge identifiers for the edges of $G \setminus \{y\}$ based on the spanning tree $\widetilde{T}$. Namely, for an edge $e = (u, v)$ of $G \setminus \{y\}$, let

$$\mathrm{EID}_{\widetilde{T}}(e) = [\mathrm{UID}(e), \mathrm{ID}(u), \mathrm{ID}(v), \mathsf{ANC}_{\widetilde{T}}(u), \mathsf{ANC}_{\widetilde{T}}(v)].$$

Now, for every vertex $v \in V \setminus \{y\}$ we define its sketch $\mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(v)$ similarly to $\mathsf{Sketch}_G(v)$, only ignoring edges incident to y in the sampling, and using the $\mathrm{EID}_{\widetilde{T}}$ identifiers for the edges. By this point of the algorithm, computing these new sketches requires $\widetilde{O}(1)$ rounds of communication, in which every $v \in C_1 \cup \cdots \cup C_k$ sends $\mathsf{ANC}_{\widetilde{T}}(v)$ to all its $(G \setminus \{y\})$-neighbors. As the T- and $\widetilde{T}$-ancestry labels coincide on the vertices of C_0, every vertex $v \in V \setminus \{y\}$ can now determine $\mathrm{EID}_{\widetilde{T}}(e)$ for every edge e incident to it in $G \setminus \{y\}$, and use the random seed $\mathcal{S}_h$ to compute $\mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(v)$.

3.1: Computing $\widetilde{T}$-Subtree Sketches. Our next goal is for every $x \in C_1 \cup \cdots \cup C_k$ to learn the $(G \setminus \{y\})$-sketch of its $\widetilde{T}$-subtree (not T-subtree), namely $\mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(V(\widetilde{T}_x)) = \oplus_{v\in\widetilde{T}_x}\mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(v)$. This is done by using y as a coordinator similarly to the $\widetilde{T}$-subtree sum computation of Step 2.1. We start by bottom-up XOR-aggregation of the sketches on each T_{y_i} (in parallel), which produces the component sketches $\mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(C_i)$. Next, within $\widetilde{O}(1)$ rounds, the component sketches are all passed to y from its children. Observe that now y can locally compute the $\widetilde{T}$-subtree sketch of each r_i as follows: $\mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(V(\widetilde{T}_{r_i})) = \oplus_{j\in J(i)}\mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(C_j)$ where $J(i)$ is the set of all indices j such that C_j is the subtree of C_i in the component tree $\widetilde{CT}$. Then y sends each of its children

y_i the $\widetilde{T}$-subtree sketch of r_i, and this information is then propagated down on each T_{y_i} (in parallel), so that each r_i learns its $\widetilde{T}$-subtree sketch. The r_i's then send their $\widetilde{T}$-subtree sketches to their $\widetilde{T}$-parent, which are the p_i's. For each vertex v of $\widetilde{T}$, let

$$\beta_v = \begin{cases} \text{if } v = p_j: & \mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(v) + \mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(V(\widetilde{T}_{r_j})) \\ \text{otherwise:} & \mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(v) \end{cases}$$

Then by this point of the algorithm, every $v \in C_1 \cup \cdots \cup C_k$ know its β_v value. For $i = 1, \ldots, k$, let $\widetilde{T}^{(i)}$ be the tree induced on C_i by $\widetilde{T}$, where the parents in $\widetilde{T}^{(i)}$ are the same as in $\widetilde{T}$. Equivalently, $\widetilde{T}^{(i)}$ is the tree obtained by rerooting T_{y_i} at the vertex r_i. Each of its leaves is either an original $\widetilde{T}$-leaf or a p_j vertex for some j. The crux is that for each $x \in C_i$ it holds that $\mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(x) = \oplus_{v\in\widetilde{T}^{(i)}_x}\beta_v$. That is, the $\widetilde{T}$-subtree sketch of x is equal to the sum-of-β's in its $\widetilde{T}^{(i)}$-subtree. Hence, we complete the computation in this step by executing bottom-up XOR-aggregation of the β_v values in each of the trees $\widetilde{T}^{(i)}$ in parallel.

3.2: Computing the Sketch of $V \setminus \{y\}$. The last required sketch ingredient for the local simulation of Borůvka in the next step is letting all vertices $x \in C_1 \cup \cdots \cup C_k$ to learn the global sum-of-sketches in $G \setminus \{y\}$, i.e. $\mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(V \setminus \{y\})$. To this end, we carefully examine the contribution of the vertices in C_0 to this sum, as some of them are not $V(T_y)$-adjacent and cannot participate in the computation. This enables us to transform the global sketch $\mathsf{Sketch}_G(V)$ (known from preprocessing) to the desired global sketch in $G \setminus \{y\}$. The details appear in the full version of the paper, in the proof of the following claim:

▷ Claim 12. In $\widetilde{O}(D)$-rounds of computation with $\widetilde{O}(1)$ congestion, in which messages are sent only on $V(T_y)$-incident messages, each vertex $x \in C_1 \cup \cdots \cup C_k$ can learn $\mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(V \setminus \{y\})$.

Step 4: Local Borůvka Simulation In $G \setminus \{x, y\}$. This entire step is executed by local computation in which each $x \in C_1 \cup \cdots \cup C_k$ determines whether it is a cut vertex in $G \setminus \{y\}$, or equivalently if xy is a cut pair in G. This is done by locally simulating Borůvka's algorithms using the sketches of the components of $\widetilde{T} \setminus \{x\}$ (which are known to x by Step 3) in an identical manner to the last step of the (single) cut vertex detection algorithm of Section A, replacing G and T there with $G \setminus \{y\}$ and $\widetilde{T}$. We note that the new ancestry labels, extended identifiers and sketches, *computed with respect to* $\widetilde{T}$, are important for this simulation to follow through exactly as in Section A. This completes the proof of Lemma 10.

We conclude this section by describing the scheduling of the algorithms $\{A_y\}_{y\in V}$:

▶ **Lemma 13.** *The collection of algorithms $\{\mathcal{A}_y\}_{y\in V}$ can be executed simultaneously within $\widetilde{O}(D)$ rounds, w.h.p.*

Proof. The key observation is that every edge e participates in $O(D)$ algorithms. Specifically, since each algorithm $\mathcal{A}_y$ exchanges messages only on edges incident to $V(T_y)$, we get that the algorithms using $e = (u, v)$ are exactly $\{\mathcal{A}_y \mid y \in \pi(s, u, T) \cup \pi(s, v, T)\}$. Therefore, the total number of messages sent through $e = (u, v)$ in the collection of n algorithms $\{A_y\}_{y\in V}$ is at most $\widetilde{O}(1) \cdot (|\pi(s, u, T)| + |\pi(s, v, T)|) = \widetilde{O}(D)$. The proof follows by employing Theorem 9 with congestion and dilation bounds of $\widetilde{O}(D)$. ◀

3 Independent Cut Pairs

We now turn to consider the case where the cut pair xy is independent, i.e., x, y have no ancestor-descendant relations. Throughout this section, for every vertex $x \in V$, let $V_x = V(T_x) \setminus \{x\}$. Recall that we assume that there is no single cut vertex in the graph. Our algorithm is based on the introduced notion of *x-connectivity trees*, $\widehat{T}_x$, computed locally at each vertex x. Let $\mathcal{C}_x = \{C_1, \ldots, C_k\}$ denote the maximal connected components in the induced graph $G[V_x]$. For each $C \in \mathcal{C}_x$, the tree $\widehat{T}_x$ contains a path $\pi_x(s, C) = \pi(s, u_C) \circ (u_C, v_C)$, where (u_C, v_C) is a G-edge such that $v_C \in C$, and $x \notin \pi_x(s, C)$. Therefore, $\widehat{T}_x$ encodes the connectivity of s to V_x in the graph $G \setminus \{x\}$. We next describe the computation of these $\widehat{T}_x$ trees, and later on show how they guide the identification of independent cut pairs. Throughout, we assume that the ID of each vertex v contains also its compressed-path information $\pi^*(s, v)$. For every $v \in V_x$, let $C_{x,v}$ denote the component containing v in $\mathcal{C}_x$. When $v = x_h$, we let $H_x = C_{x,x_h}$ and denote it as the *heavy component* of x.

3.1 Computing x-Connectivity Trees

The computation has two main steps, both are based on the bottom-up aggregation of certain graph sketches over the BFS tree T. The purpose of first step is to allow every $x \in V$ to determine the connected components $\mathcal{C}_x$ in $G[V_x]$ where each such component C is identified by the vertex of largest ID among all the T-children of x in C. In addition, in the output of this step each vertex $u \in V_x$ learns the ID of its component $C_{x,u} \in \mathcal{C}_x$. The second step aggregates a special form of graph sketches that provide x with the required path information in order locally compute $\widehat{T}_x$.

Step 1: Computing Connectivity in $G[V_x]$. For ease of notation, let $D = \text{depth}(T)$ and $d_x = \text{depth}(x)$ denote the depth of x in T. We say that an edge $e = (u, v)$ has depth d if $\text{depth}(\text{LCA}(u, v)) = d$. To locally simulate the connectivity Borůvka algorithm in $G[V_x]$ at every x, it is required for x to learn $\mathsf{Sketch}_{G[V_x]}(V(T_w))$ for each T-child w of x. Observe that the edges of $G[V_x]$ can be identified as G-edges in $V_x \times (V \setminus \{x\})$ of depth at least d_x. For this purpose, the algorithm is based on aggregating the information of D graph sketches, for every depth $d \in \{1, \ldots, D\}$. The computation of dth sketch $\mathsf{Sketch}^d_G(\cdot)$ will be restricted to sampling only edges of depth *at least* d. We thus obtain the following lemma. Missing proofs in this section appear in the full version of the paper.

▶ **Lemma 14.** *There is a randomized $\widetilde{O}(D)$-round algorithm that w.h.p. computes connectivity in each $G[V_x]$ for every $x \in V$ simultaneously. At the end of the execution, each u holds a component-ID in the graph $G[V_x]$ for every $x \in \pi(s, u)$. Moreover, within additional $\widetilde{O}(D)$ rounds, each u can send its entire component-ID information (for every $x \in \pi(s, u)$) to all its neighbors.*

Step 2: Computing x-Connectivity Trees $\widehat{T}_x$ via Path-Sketches. Our next goal is to provide each vertex x with the path information $\pi_x(s, C)$, for every component $C \in \mathcal{C}_x$. Such a path connects a vertex $v_C \in C$ to the source s in $G \setminus \{x\}$. As we assume that x is not a cut vertex, such a path indeed exists. Towards that goal, we augment the identifier of each edge (u, v) with the tree paths $\pi(s, u), \pi(s, v)$. Formally,

$$\text{EID}^P_T(e) = [\text{UID}(e), \text{ID}(u), \text{ID}(v), \mathsf{ANC}_T(u), \mathsf{ANC}_T(v), \pi(s, u), \pi(s, v)] \; . \tag{2}$$

In contrast to the extended-ID of Eq. (1) which have $\widetilde{O}(1)$ bits, the latter $\text{EID}^P_T(e)$ identifiers have $\widetilde{O}(D)$ bits. The sketches obtained with these $\text{EID}^P_T(e)$ IDs are called *path-sketches*, denoted as $\mathsf{Sketch}^P_G(S)$ for $S \subseteq V$. The advantage of these path-sketches is that any detected

outgoing edge (u, v) obtained from $\mathsf{Sketch}_G^P(Q)$ includes the path information $\pi(s, u)$ and $\pi(s, v)$. Note that the path-sketches $\mathsf{Sketch}_G^P(S)$ have $\widetilde{O}(D)$ bits, since the edge IDs have now $\widetilde{O}(D)$ bits.

Our goal is to let each x learn the path-sketches $\mathsf{Sketch}_G^P(C)$ for each component $C \in \mathcal{C}_x$. Since each path-sketch has $\widetilde{O}(D)$ bits, we cannot allow to compute D sketches for each depth $d \in \{1, \ldots, D\}$. Instead we only aggregate the $\mathsf{Sketch}_G^P(u)$ information in a bottom-up manner on T, which allows every vertex x to learn $\mathsf{Sketch}_G^P(V(T_w))$ for each of its T-children w. By combining with the output of the first step, x can then determine $\mathsf{Sketch}_G^P(C)$ for every $C \in \mathcal{C}_x$.

▶ **Lemma 15.** *W.h.p., all vertices x can compute the x-connectivity trees $\widehat{T}_x$ within $\widetilde{O}(D)$ randomized rounds.*

For each $x \in V$ and $C \in \mathcal{C}_x$, we define the compressed path of $\pi_x(s, C)$ as $\pi^*_x(s, C) = \pi^*(s, v_C) \circ (v_C, u_C)$ (hence, $\pi^*_x(s, C)$ has $\widetilde{O}(1)$ bits). We conclude the computation regarding the connectivity trees by letting each vertex v learn the compressed-path $\pi^*_x(s, C_{x,v})$ for each of its ancestors $x \in \pi(s, v)$. Since the compressed-path has $\widetilde{O}(1)$ bits, a vertex is required to receive $\widetilde{O}(D)$ bits of information, which can be done in $\widetilde{O}(D)$ rounds:

▶ **Lemma 16.** *There is an $\widetilde{O}(D)$-round algorithm that allows each vertex v to learn the compressed path $\pi^*_x(s, C_{x,v})$ for each $x \in \pi(s, v)$, as well as the entire path $\pi_x(s, C_{x,v})$ for each $x \in \mathsf{LA}(v)$. In addition, each vertex v can share all of this information with neighbors.*

Proof. We let every vertex x send the full path $\pi_x(s, C_{x,x'})$ to each light child x' of x, and the compressed path $\pi^*_x(s, H_x)$ to its heavy child x_h. This information is propagated towards the leaf vertices of T_x. Since each vertex is required to receive $\widetilde{O}(D)$ bits of information from each of its *light* ancestors, as well as $\widetilde{O}(1)$ bits from each of its heavy ancestors, overall it is required to receive $\widetilde{O}(D)$ bits. This can be done in $\widetilde{O}(D)$ rounds, by standard pipeline techniques. Since each v learns $\widetilde{O}(D)$ bits of information, the learned information can be exchanged between every pair of neighbors within $\widetilde{O}(D)$ rounds, as well. ◀

3.2 Component Classification Based on Sensitivity

We next use the structure of the x-connectivity tree $\widehat{T}_x$ to classify the xy pairs into several types. We also filter-out possibly many irrelevant xy pairs (for which we deduce immediately that xy is not a cut) using the notion of *sensitivity*.

▶ **Definition 17** (Sensitivity Notions of $\mathcal{C}_x$ Components). *Fix an independent pair x, y. A component $C \in \mathcal{C}_x$ is y-*sensitive *if $y \in \pi_x(s, C)$. The y-sensitive components of $\mathcal{C}_x$ are further classified into two types:* pseudo-sensitive *and* fully-sensitive*, as follows. A component $C \in \mathcal{C}_x$ is* pseudo y-sensitive *if the tree path $\pi_x(s, C)$ contains some edge (y, y') such that $x \notin \pi_y(s, C_{y,y'})$, where $C_{y,y'}$ is the component containing y' in $\mathcal{C}_y$. Finally, a y-sensitive component $C \in \mathcal{C}_x$ is* fully y-sensitive *if C is not pseudo-sensitive.*

Hence, in particular a component $C \in \mathcal{C}_x$ is fully y-sensitive if either that last edge of $\pi_x(s, C)$ is incident to y, or that there is an edge $(y, y') \in \pi_x(s, C)$ such that the component $C_{y,y'} \in \mathcal{C}_y$ is x-sensitive. Note that non-y-sensitive components are clearly connected to s in $G \setminus \{x, y\}$. We later on show that this is true also for pseudo y-sensitive components, therefore their sensitivity to y is superficial. Let $\mathcal{S}(x, y), \mathcal{PS}(x, y), \mathcal{FS}(x, y)$ denote the components in $\mathcal{C}_x$ that are y-sensitive, pseudo y-sensitive and fully y-sensitive, respectively[7]. We next

[7] Notice that these notations are not symmetric in x, y, e.g. $\mathcal{S}(x, y)$ is different than $\mathcal{S}(y, x)$.

show that each vertex x can determine, for every $C \in \mathcal{C}_x$, certain y vertices for which C is fully y-sensitive by running the procedure described of the following lemma. Note that by having the compressed-path $\pi^*_y(s, C_{y,y'})$ and the $\pi^*(s, x)$, it is possible to determine if $x \in \pi_y(s, C_{y,y'})$, hence determining if $C_{y,y'}$ is x-sensitive.

▶ **Lemma 18.** *There is an $\widetilde{O}(D)$-round algorithm that computes the following for every $x \in V$ (in parallel):*

- *$\pi^*_y(s, C_{y,y'})$ for every edge $(y, y') \in \pi_x(s, C)$ and every $C \in \mathcal{C}_x \setminus \{H_x\}$.*
- *$\pi^*_y(s, C_{y,y'})$ for every* light *edge $(y, y') \in \pi_x(s, H_x)$.*

3.3 xy-Connectivity Algorithms Under a Promise

Throughout, we assume that all vertices applied the pre-processing steps of computing the x-connectivity trees $\widehat{T}_x$, as well as, applied the $\widetilde{O}(D)$-round procedures of Lemma 16 and 18. From this point on, we explain how to determine the connectivity in $G \setminus \{x, y\}$, first for a single pair xy, and then for all pairs that satisfy a given promise.

Recall that $\mathsf{LD}(x)$ is the collection of light descendants of x in T. For a vertex y, let $\mathsf{LDS}(x, y)$ be the collection of light descendants of x that are sensitive to y. Formally, the **l**ight x-**d**escendants y-**s**ensitive vertices are defined by:

$$\mathsf{LDS}(x, y) = \{v \in \mathsf{LD}(x) \mid y \in \pi_x(s, C_{x,v}) \setminus V(T_x)\} . \tag{3}$$

▶ **Observation 19.** *Every vertex v belongs to a total of $O(D \log n)$ sets $\mathsf{LDS}(x, y)$ for $x, y \in V$.*

Proof. A vertex $v \in V$ has $O(\log n)$ light ancestors (i.e., belongs to $O(\log n)$ sets of $\mathsf{LD}(x)$). In addition, for each light ancestor $x \in \pi(s, v)$, there are $O(D)$ vertices $y \in \pi_x(s, C_{x,v})$. Therefore, it belongs to $O(D \log n)$ sets as required. ◀

▶ **Theorem 20** (xy-Connectivity Given an x-y Path)**.** *Fix $x, y \in V$ and assume that there is an x-y path $\Pi_{x,y} \subseteq G$ (known in a distributed manner) of length $O(D)$. Then, there is an xy-connectivity algorithm $\mathcal{A}^P_{x,y}$ (i.e., that determines the connectivity in $G \setminus \{x, y\}$) in $\widetilde{O}(D)$ and $\widetilde{O}(1)$-congestion, by sending messages only along on the edges of $\Pi_{x,y}$ or edges incident to $\mathsf{LDS}(x, y) \cup \mathsf{LDS}(y, x)$. At the end of the computation, both x and y know whether $G \setminus \{x, y\}$ is connected or not.*

Before proving Theorem 20, we show that given a set of pairs $Q \subseteq V \times V$, then all algorithms $\{\mathcal{A}^P_{x,y} \mid (x, y) \in Q\}$ can be scheduled simultaneously when provided a path collection $\mathcal{P}_Q = \{\Pi_{x,y} \mid (x, y) \in Q\}$ that satisfies the following promise:

[**Promise**:] *$\mathcal{P}_Q$-paths have length $O(D)$, and each edge appears on $\widetilde{O}(D)$ paths in $\mathcal{P}_Q$.*

By a straightforward application of the random delay approach, we obtain:

▶ **Corollary 21.** *[All Pairs xy-Connectivity Under a Promise] Let $Q \subseteq V \times V$ be a collection of independent pairs and let $\mathcal{P}_Q = \{P_{x,y} \mid (x, y) \in Q\}$ be a collection of x-y paths that satisfy the promise. Then, the collection of algorithms $\{\mathcal{A}^P_{x,y} \mid (x, y) \in Q\}$, where each $\mathcal{A}_{x,y}$ uses the corresponding path $\Pi_{x,y} \in \mathcal{P}_Q$, can be scheduled simultaneously within $\widetilde{O}(D)$ rounds, w.h.p.*

Description of the Connectivity Algorithm $\mathcal{A}^P_{x,y}$. The algorithm is based on simulating the Borůvka algorithm using the sketch information of connected subsets in $G \setminus \{x, y\}$, held jointly by x and y. Throughout, we refer to the given x-y path $\Pi_{x,y}$ as *the xy channel*. Recall that the algorithm can send only $\widetilde{O}(1)$ bits on that channel. The input for the $i \geq 1$

phase of Borůvka is the following. There is a partitioning $\mathcal{P}_{i-1} = \{P_{i-1,1}, \ldots, P_{i-1,k_{i-1}}\}$ of the vertices in $V \setminus \{x, y\}$ into connected subsets (in $G \setminus \{x, y\}$). We call each $P \in \mathcal{P}_{i-1}$ a *part* (to avoid confusion with the term "component" reserved for sets in $\mathcal{C}_x$ and $\mathcal{C}_y$). We mark a special vertex in each $P_{i,j} \in \mathcal{P}_i$, called the *leader* of the part. The source vertex s is the leader of its own part (called the s-part), and the leaders of the other parts are some chosen T-children of x or y in these parts. The part-ID is the ID of its leader. The part containing x_h (resp., y_h) is called *x-heavy* (resp., *y-heavy*)[8]. The parts that are free of s, x_h, y_h are called *light.* Hence every light part is contained in $\mathsf{LD}(x) \cup \mathsf{LD}(y)$. A part P is denoted as *growable* if there is an outgoing G-edge connecting P to $V \setminus (P \cup \{x, y\})$. The Borůvka algorithm has $K = O(\log n)$ forest growing phases in $G \setminus \{x, y\}$, each phase reduces the number of growable parts by a constant factor, in expectation. We maintain the following invariant for the beginning of each phase $i \in \{1, \ldots, K\}$:

(I1) x, y know $\mathsf{Sketch}_{G\setminus\{x,y\}}(P)$ of the part $P \in \mathcal{P}_{i-1}$ containing s.
(I2) $z \in \{x, y\}$ knows $\mathsf{Sketch}_{G\setminus\{x,y\}}(P)$ for every *light* part $P \in \mathcal{P}_{i-1}$ whose leader is in T_z.
(I3) x, y know $\mathsf{Sketch}_{G\setminus\{x,y\}}(P)$ as well as the part-IDs of the heavy parts P in $\mathcal{P}_{i-1}$.
(I4) $z \in \{x, y\}$ knows, for each T-child z' of z, the part-ID of the part containing z' in $\mathcal{P}_{i-1}$.

Satisfying the Invariant for the First Borůvka Phase. We start by defining the partitioning $\mathcal{P}_0$ and in particular, focus first on the definition of the part containing s. Recall Def. 17 and that $\mathcal{S}(x, y), \mathcal{PS}(x, y), \mathcal{FS}(x, y) \subseteq \mathcal{C}_x$ are the y-sensitive, pseudo y-sensitive and fully y-sensitive components, respectively. Let $\mathsf{NS}(x, y) = \bigcup_{C \in \mathcal{C}_x \setminus \mathcal{FS}(x,y)} C$. The set $\mathsf{NS}(y, x)$ is defined in an analogous manner. Then the s-part in $\mathcal{P}_0$ is given by $U(x, y) = (V \setminus (V(T_x) \cup (T_y))) \cup \mathsf{NS}(x, y) \cup \mathsf{NS}(y, x)$. The next observation exploits the fact that the pseudo y-sensitive components in $\mathcal{C}_x$ and the pseudo x-sensitive components in $\mathcal{C}_y$ are all connected to s in $G \setminus \{x, y\}$.

▶ **Observation 22.** *$G[U(x, y)]$ is connected.*

We partition the responsibilities on the parts in $\mathcal{P}_0$ between x and y, as follows. Let $\mathcal{P}_{0,x} = \mathcal{FS}(x, y)$ be the components in $\mathcal{C}_x$ that are fully-sensitive to y. Similarly, $\mathcal{P}_{0,y} = \mathcal{FS}(y, x)$. The 0th partitioning of $V \setminus \{x, y\}$ is given by $\mathcal{P}_0 = \{U(x, y)\} \cup \mathcal{P}_{0,x} \cup \mathcal{P}_{0,y}$. For every $z \in \{x, y\}$, the leader of each $C \in \mathcal{P}_{0,z}$ is chosen as the vertex of largest ID among all the T-children of x, y in C. The leader of $U(x, y)$ is the root s. To satisfy the invariants for the beginning of phase $i \geq 1$, it is sufficient to show the following claims for x (as they apply in a symmetric manner also for y):

▷ Claim 23. Within $\widetilde{O}(D)$ rounds, the vertex x can compute $\mathsf{Sketch}_{G\setminus\{x,y\}}(C)$ for every component $C \in \mathcal{S}(x, y)$. In addition, the vertex y can determine its neighbors in $\{v \in V_x \mid y \notin \pi_x(s, C_{x,v})\}$. The communication is restricted to the edges of $\mathsf{LDS}(x, y) \cup \mathsf{LDS}(y, x)$ and using the xy channel.

▷ Claim 24. By exchanging $\widetilde{O}(1)$ bits of information (using the promised channel), invariants (I1-I4) hold w.r.t $\mathcal{P}_0$.

Simulation of the ith Borůvka Phase. We now describe the execution of phase $i \geq 1$ assuming that at the beginning of the phase the invariant holds w.r.t $\mathcal{P}_{i-1}$. The output of the execution will be the partitioning $\mathcal{P}_i$, for which we later show that the invariant holds

[8] A part can be both x-heavy and y-heavy.

as well. Our goal is to let x, y simulate a Borůvka phase in which parts of $\mathcal{P}_{i-1}$ are merged along their outgoing edges. See Figure 2 in Appendix B for an illustration of this process. The main objective of this phase is to reduce the number of *growable* parts by a constant factor, in expectation. Throughout, we use the following auxiliary claim which allows the vertices in every light part to exchange $\widetilde{O}(1)$ bits, in parallel.

▷ Claim 25. Let P be a light part in $\mathcal{P}_{i-1}$ such that each vertex $v \in P$ holds a $\widetilde{O}(1)$-bit value $val(v)$. Then, there is an $\widetilde{O}(D)$-round algorithm that allows all vertices in P to compute any aggregate function of the $val(v)$ values for $v \in P$, by sending messages *only* along edges incident to P. Consequently, all light parts in $\mathcal{P}_{i-1,x} \cup \mathcal{P}_{i-1,y}$ can compute their respective aggregate functions, in parallel.

For efficiency of computation, we restrict the merge shapes to be star shapes by using random coins (see e.g., [15]). Such star merges are obtained by letting each part $\mathcal{P}_{i-1}$ toss a random coin, and allowing only merges centered on head-parts, each accepting incoming suggested merge-edges from tail-parts. The leader of this head-part becomes the leader of the merged part. We show that under the promise and the $(i-1)^{th}$ invariant, this merging phase can be implemented in $\widetilde{O}(D)$ rounds as follows. W.l.o.g., we make x be responsible for the s-part $P_s \in \mathcal{P}_{i-1}$.

Implementing Merges. Each vertex $z \in \{x, y\}$ tosses a (fresh) random coin for each of its parts in $\mathcal{P}_{i-1,z}$. In addition, x tosses a coin for the s-part P_s. Next, for each of the tail part $P \in \mathcal{P}_{i-1,z}$, z locally computes an outgoing edge for each of its *tail* parts in $\mathcal{P}_{i-1,z}$. In addition, x computes an outgoing edge for the s-part (in case that the coin toss of that part is tail). For each growable part $P \in \mathcal{P}_{i-1,z}$, such an edge can be detected from $\mathsf{Sketch}_{G \setminus \{x,y\}}(P)$ with constant probability. The parts of $\mathcal{P}_i$ are formed by merging every head part $P^* \in \mathcal{P}_{i-1}$ with all the tail parts in $\mathcal{P}_{i-1}$ whose outgoing edges point at P^*. The leader of the merged part is the leader of the head part P^*. For every tail part $P \in \mathcal{P}_{i-1,x}$, let $e_P = (u_P, v_P)$ be the detected outgoing edge obtained by x from $\mathsf{Sketch}_{G \setminus \{x,y\}}(P)$.

▷ Claim 26. Using $\widetilde{O}(D)$ rounds of communication over edges incident to $\mathsf{LDS}(x, y)$ and the given xy channel, z can determine for all its tail parts $P \in \mathcal{P}_{i-1,z}$ with an outgoing edge $e_P = (u_P, v_P)$, the following information: (i) the part-ID of the second endpoint $v_P \notin P$ and, (ii) the coin-toss of the part of v_P.

To implement the merges and satisfy the invariant, it is required for $z \in \{x, y\}$ to learn the updated sketch information of their head parts in $\mathcal{P}_{i-1,z}$. We next explain how y can compute the sketch information of each of its head parts $P^* \in \mathcal{P}_{i-1,y}$. (A similar procedure would work for x).

By Claim 26, x knows for every head part $P^* \in \mathcal{P}_{i-1,y}$, the collection of tail parts in $\mathcal{P}_{i-1,x}$ that should be merged with P^*.

Any $\rightarrow$ Non-Light Merges. There are (at most three) non-light parts in $\mathcal{P}_{i-1}$, corresponding to at most two heavy parts and the s-part[9]. For each of these non-light part P^*, x aggregates that sketch information of the corresponding tail parts $P \in \mathcal{P}_{i-1,x}$, and send it to y over the xy channel.

From the point on, x considers the transfer of information concerning the light head parts P^* in $\mathcal{P}_{i-1,y}$.

[9] The latter is held by x, so when revering the roles of x, y, y might be required to send x the sum of sketch information of the tail parts in $\mathcal{P}_{i-1,y}$ that got merged with the s-part.

Non-Light → Light Merges. It uses the xy channel to send y the sketch information of its non-light tail parts P, along with the part-ID of their head parts (to which they should be merged).

Light → Light Merges. The sketch of all other (light) parts in $\mathcal{P}_{i-1,x}$ are communicated to y over the edges incident to the light sensitive xy descendants, $\mathsf{LDS}(x, y) \cup \mathsf{LDS}(y, x)$, as follows. Using Claim 25, each light and tail part $P \in \mathcal{P}_{i-1,x}$ can learn $\mathsf{Sketch}_{G\setminus\{x,y\}}(P)$ (as x holds this information, by the invariant). Note that by definition $P, P^* \subseteq \mathsf{LDS}(x, y) \cup \mathsf{LDS}(y, x)$. The vertices of P then send this received information to all their neighbors. At this point, for every light head part P^* in $\mathcal{P}_{i-1,y}$, and for every tail *light* part P in $\mathcal{P}_{i-1,x}$, there is a vertex $v_P \in P^*$ that holds $\mathsf{Sketch}_{G\setminus\{x,y\}}(P)$. By applying Claim 25, all vertices in P^* can learn the sum of all these sketches. This provides y with all the required information from x to compute $\mathsf{Sketch}_{G\setminus\{x,y\}}(P^*)$ for each head part $P^* \in \mathcal{P}_{i-1,y}$. In a symmetric manner, x can compute the sketch of the merged parts for all its head parts in $\mathcal{P}_{i-1,x}$. Using the xy channel, x and y can exchange the part-ID and sketch information of the heavy parts and the s-part in $\mathcal{P}_i$. This satisfies (I1,I2,I3) for the partitioning $\mathcal{P}_i$.

To satisfy (I4), note that the part-ID has changed only for tail parts in $\mathcal{P}_{i-1}$. For the tail-parts in $\mathcal{P}_{i-1,z}$, z holds their new part-ID using Claim 26 (i.e., this is the part-ID of the detected outgoing edges). This completes the description for phase i.

We are now ready to complete the proof of Theorem 20.

Proof of Theorem 20. By the description of the ith phase, the invariant holds w.r.t $\mathcal{P}_i$. We next show that the ith phase sends $\widetilde{O}(1)$ messages along edges incident to $\mathsf{LDS}(x, y) \cup \mathsf{LDS}(y, x)$, as well as over the xy channel. It is also easy to see that given the promised channel that running time is $\widetilde{O}(D)$ using Claim 25. Finally, we show that within $k = O(\log n)$ phases it holds that there are no growable components in $\mathcal{P}_k$.

Recall that a part P in $\mathcal{P}_j$ is denoted as *growable* if there is a G-edge $(u, v) \in P \times (V \setminus (\{x, y\} \cup P))$. We claim that the number of growable part reduces by a constant factor in each Borůvka phase. Given a sketch information $\mathsf{Sketch}_{G\setminus\{x,y\}}(P)$ for a growable part P, one can infer an outgoing edge (u, v) from P with constant probability. In addition, with probability 1/4 this edge is valid (i.e., P is a tail part and v is in a head part). Therefore, overall the number of growable parts reduces by a constant factor, in expectation. By the Markov inequality, w.h.p. there is no growable part after $O(\log n)$ phases. Since x, y jointly hold the sketch information of all parts in $\mathcal{P}_k$ they can determine if there is more than one part in $\mathcal{P}_k$ by exchanging information along their channel (i.e., if $G \setminus \{x, y\}$ is not connected, then w.h.p. either x or y holds a part whose leader is not s). The theorem follows. ◀

References

1 Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 459–467. SIAM, 2012.

2 Giuseppe Di Battista and Roberto Tamassia. Incremental planarity testing (extended abstract). In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October – 1 November 1989*, pages 436–441. IEEE Computer Society, 1989.

3 Giuseppe Di Battista and Roberto Tamassia. On-line maintenance of triconnected components with spqr-trees. *Algorithmica*, 15(4):302–318, 1996.

4 Keren Censor-Hillel, Mohsen Ghaffari, and Fabian Kuhn. Distributed connectivity decomposition. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 156–165. ACM, 2014.

5 Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. *CoRR*, abs/2203.00671, 2022. `doi:10.48550/arXiv.2203.00671`.

6 Keerti Choudhary. An optimal dual fault tolerant reachability oracle. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPIcs*, pages 130:1–130:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016.

7 Michal Dory, Yuval Efron, Sagnik Mukhopadhyay, and Danupon Nanongkai. Distributed weighted min-cut in nearly-optimal time. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 1144–1153. ACM, 2021.

8 Michal Dory and Merav Parter. Fault-tolerant labeling and compact routing schemes. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 445–455. ACM, 2021.

9 Ran Duan and Seth Pettie. Dual-failure distance and connectivity oracles. In *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*, pages 506–515. SIAM, 2009.

10 Ran Duan and Seth Pettie. Connectivity oracles for graphs subject to vertex failures. *CoRR*, abs/1607.06865, 2016. `arXiv:1607.06865`.

11 Ran Duan and Seth Pettie. Connectivity oracles for graphs subject to vertex failures. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 490–509, 2017.

12 Pawel Gawrychowski, Shay Mozes, and Oren Weimann. Minimum cut in $o(m \log^2 n)$ time. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 57:1–57:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

13 Loukas Georgiadis, Giuseppe F. Italiano, Luigi Laura, and Nikos Parotsidis. 2-vertex connectivity in directed graphs. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming – 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, volume 9134 of *Lecture Notes in Computer Science*, pages 605–616. Springer, 2015.

14 Mohsen Ghaffari. Near-optimal scheduling of distributed algorithms. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC*, pages 3–12, 2015.

15 Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks II: low-congestion shortcuts, MST, and min-cut. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 202–219. SIAM, 2016.

16 Mohsen Ghaffari, Krzysztof Nowicki, and Mikkel Thorup. Faster algorithms for edge connectivity via random 2-out contractions. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 1260–1279. SIAM, 2020.

17 Mohsen Ghaffari and Merav Parter. MST in log-star rounds of congested clique. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 19–28, 2016.

18 David Gibb, Bruce M. Kapron, Valerie King, and Nolan Thorn. Dynamic graph connectivity with improved worst case update time and sublinear space. *CoRR*, abs/1509.06464, 2015. `arXiv:1509.06464`.

19 Manoj Gupta and Shahbaz Khan. Multiple source dual fault tolerant BFS trees. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 127:1–127:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017.

20 Zhiyang He, Jason Li, and Magnus Wahlström. Near-linear-time, optimal vertex cut sparsifiers in directed acyclic graphs. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 52:1–52:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

21 John E. Hopcroft and Robert Endre Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2(3):135–158, 1973.

22 Michael Kapralov and David Woodruff. Spanners and sparsifiers in dynamic streams. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 272–281, 2014.

23 Bruce M Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pages 1131–1142. SIAM, 2013.

24 David R Karger. Random sampling in cut, flow, and network design problems. *Mathematics of Operations Research*, 24(2):383–413, 1999.

25 Karthik C. S. and Merav Parter. Deterministic replacement path covering. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10–13, 2021*, pages 704–723. SIAM, 2021.

26 Valerie King, Shay Kutten, and Mikkel Thorup. Construction and impromptu repair of an MST in a distributed network with o(m) communication. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21–23, 2015*, pages 71–80, 2015.

27 Frank Thomson Leighton, Bruce M Maggs, and Satish B Rao. Packet routing and job-shop scheduling ino (congestion+ dilation) steps. *Combinatorica*, 14(2):167–186, 1994.

28 Jason Li, Danupon Nanongkai, Debmalya Panigrahi, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Vertex connectivity in poly-logarithmic max-flows. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 317–329. ACM, 2021.

29 Ali Mashreghi and Valerie King. Broadcast and minimum spanning tree with o(m) messages in the asynchronous CONGEST model. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, pages 37:1–37:17, 2018.

30 Danupon Nanongkai, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Breaking quadratic time for small vertex connectivity and an approximation scheme. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 241–252. ACM, 2019.

31 Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. Otakar Borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233(1):3–36, 2001.

32 Merav Parter. Dual failure resilient BFS structure. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 481–490, 2015.

33 Merav Parter. Small cuts and connectivity certificates: A fault tolerant approach. In *33rd International Symposium on Distributed Computing*, 2019.

34 Merav Parter. Distributed constructions of dual-failure fault-tolerant distance preservers. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 21:1–21:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

35 David Peleg. *Distributed Computing: A Locality-sensitive Approach.* SIAM, 2000.

36 Seth Pettie and Longhui Yin. The structure of minimum vertex cuts. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 105:1–105:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

37 David Pritchard and Ramakrishna Thurimella. Fast computation of small cuts via cycle space sampling. *ACM Transactions on Algorithms (TALG)*, 7(4):46, 2011.

38 Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

39 Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

40 Ramakrishna Thurimella. Sub-linear distributed algorithms for sparse certificates and biconnected components. *Journal of Algorithms*, 23(1):160–179, 1997.

41 Salil P. Vadhan. Pseudorandomness. *Foundations and Trends® in Theoretical Computer Science*, 7(1–3):1–336, 2012. doi:10.1561/0400000010.

42 Oren Weimann and Raphael Yuster. Replacement paths and distance sensitivity oracles via fast matrix multiplication. *ACM Transactions on Algorithms (TALG)*, 9(2):14, 2013.

A Single Cut Vertices

In this section we describe the distributed algorithm for detecting single vertex cuts of Theorem 1. This serves both as a warm-up to our approach in the subsequent sections devoted to dual vertex cuts detection, as well as for a detailed presentation of basic tools used in these next sections. We assume each vertex v is equipped with its heavy/light classification in T and with its ancestry label which is its compressed path, $\mathsf{ANC}_T(v) = \pi^*(s, v, T)$. This can be achieved in $\widetilde{O}(D)$ rounds by Lemma 5.

Step 0: Computing Extended Edge IDs. The source s samples a random seed $\mathcal{S}_{ID}$ of $\widetilde{O}(1)$ bits and shares it with all vertices. Then, using Lemma 6, each vertex v can then locally compute the unique edge-ID $\mathrm{UID}(e)$ for each of its incident edges. By letting all neighbors in G exchange their ANC_T-labels, each $\mathrm{UID}(e)$ can be concatenated with the required information to create $\mathrm{EID}(e)$.

Step 1: Computing Subtree Sketches. The source s locally samples the random seed $\mathcal{S}_h$ of $\widetilde{O}(1)$ bits and sends it to all the vertices. Along with the extended edge IDs, this provides all the required information for the computation of $\mathsf{Sketch}_G(v)$ locally in each vertex v. By XOR-aggregation of the individual sketches from the leaves of T up to the root s, each vertex v obtains its subtree sketch, given by $\mathsf{Sketch}_G(V(T_v)) = \oplus_{u \in T_v} \mathsf{Sketch}_G(v)$. Next, within $\widetilde{O}(1)$ rounds, each vertex passes its subtree sketch to its parent, so that each vertex now holds the subtree sketch for each of its children. Finally, the source s also broadcasts its subtree sketch, which is $\mathsf{Sketch}_G(V)$, to all the other vertices.

Step 2: Local Connectivity Computation. This step is locally applied at every vertex x, and requires no additional communication. We show that each vertex x, given the received sketch information in Step 1, can locally simulate the Borůvka's algorithm [31] in the graph $G \setminus \{x\}$, and consequently determine if $G \setminus \{x\}$ is connected. Let $x_1, \ldots, x_k$ be the children of x in T. We assume that $x \neq s$; the case $x = s$ is easier and requires only slight modifications. The connected components in $T \setminus \{x\}$ are denoted by $\mathcal{C}_x = \{V(T_{x_j}) \mid j = 1, \ldots, k\} \cup \{V \setminus V(T_x)\}$. By Step 1, x holds the G-sketch of each component in $\mathcal{C}_x$: It

has explicitly received $\mathsf{Sketch}_G(V(T_{x_j}))$ from each child x_j. In addition, it can locally infer $\mathsf{Sketch}(V \setminus V(T_x)) = \mathsf{Sketch}(V) \oplus \mathsf{Sketch}(V(T_x))$. To implement Borůvka's algorithm on these components, we first need to update these G-sketches into $(G \setminus \{x\})$-sketches.

2.1: Obtaining Sketch Information in $G \setminus \{x\}$. Recall x knows the random seed $\mathcal{S}_h$ as well as the extended identifiers of its incident edges (from Step 0). For each such edge (x, u), it first uses the ancestry label of u and of its T-children (found in the EID_T's) to determine the component C of u in $\mathcal{C}_x$. It then cancel this edges from the sketch of the component C using Lemma 8. This allows x to obtain $\mathsf{Sketch}_{G\setminus\{x\}}(C)$ for every $C \in \mathcal{C}_x$.

2.2: Simulating Borůvka in $G \setminus \{x\}$. The input to this step is the $(G \setminus \{x\})$-sketch information of the components in $\mathcal{C}_{x,0} = \mathcal{C}_x$. The desired output is determining the connectivity of $G \setminus \{x\}$. The algorithm consists of $O(\log n)$ phases of the Borůvka algorithm, and is very similar to the (centralized) decoding algorithm of [8]. Each phase i will be given as input a partitioning $\mathcal{C}_{x,i} = \{C_{i,1}, \ldots, C_{i,k_i}\}$ of (not necessarily maximal) connected components in $G \setminus \{x\}$ along with their sketch information $\mathsf{Sketch}_{G\setminus\{x\}}(C_{i,j})$. The output of the phase is a coarser partitioning $\mathcal{C}_{x,i+1}$, along with the sketch information of the new parts. A component $C_{i,j} \in \mathcal{C}_{x,i}$ is said to be *growable* if it has at least one outgoing edge to a vertex in $V \setminus (C_{i,j} \cup \{x\})$. To obtain outgoings edges from the growable components in $\mathcal{C}_{x,i}$, the algorithm uses the ith basic-unit sketch $\mathsf{Sketch}_{G\setminus\{x\},i}(C_{i,j})$ of each $C_{i,j} \in \mathcal{C}_{x,i}$. By Lemma 7, from every growable component $C_{i,j} \in \mathcal{C}_{x,i}$, we get one outgoing edge $e = (u, v)$ with constant probability. To find the component $C_{i,j'}$ containing the other endpoint of e (to be merged with $C_{i,j}$), we use the T-ancestry labels found in $\mathrm{EID}_T(e)$. Say this endpoint is v. We determine the component of v in $T \setminus \{x\}$, i.e. the component $C_{0,q}$ containing v in $\mathcal{C}_{x,0}$, by querying the ancestry relation between v and each child of x using $\mathsf{ANC}_T(v)$ and the labels of x's children. Then v belongs to the unique component $C_{i,j'} \in \mathcal{C}_{x,i}$ containing $C_{0,q}$. The sketch information for the next phase $i + 1$ is given by XORing over the sketches of the components in $\mathcal{C}_{x,i}$ that got merged into a single component in $\mathcal{C}_{x,i+1}$. Note that it is important to use fresh randomness (i.e., independent sketch information) in each of the Borůvka phases [1, 23, 10]. Since each growable component gets merged with constant probability, the expected number of growable components is reduced by a constant factor in each phase. Thus after $O(\log n)$ phases, the expected number of growable components is at most $1/n^5$, and by Markov's inequality we conclude that w.h.p. there are no growable components. The partitioning at this point corresponds to the maximal connected components in $G \setminus \{x\}$, so its connectivity can be inferred. This concludes the proof of Theorem 1.

Finally, we note that by tracking the merges throughout the Borůvka simulation, x can also find a subset $\widetilde{E}$ of the outgoing edges received throughout the simulation such $(T \setminus \{x\}) \cup \widetilde{E}$ is a maximal spanning forest of $G \setminus \{x\}$.

B Figures

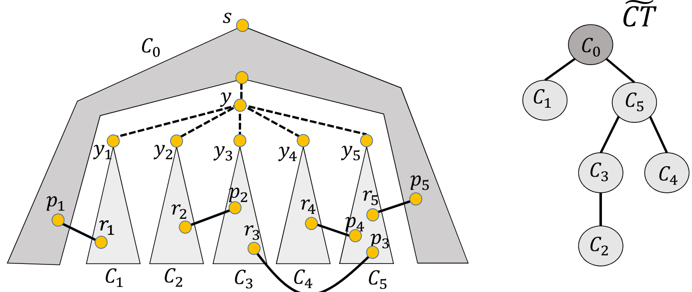


Figure 1 Left: Illustration of the trees T and $\widetilde{T}$. The dashed edges are T-edges adjacent to y, and the solid edges are $\widetilde{E}$-edges. The components $C_0, C_1, \ldots, C_5$ are each internally connected via original T-edges. The tree $\widetilde{T}$ is obtained by removing y and its incident edges from the T and adding the $\widetilde{E}$ edges. Right: The component tree $\widetilde{CT}$.

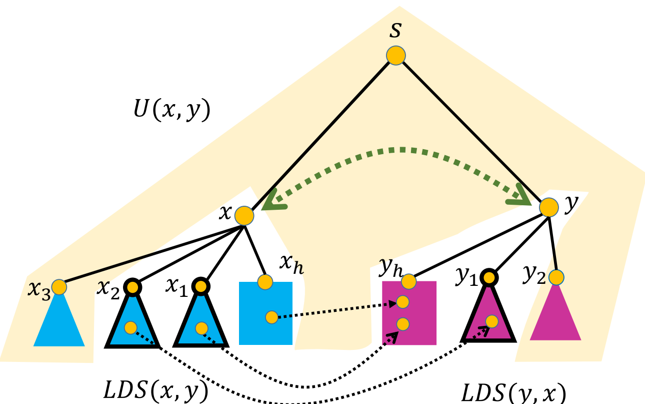


Figure 2 Simulating the first Borůvka phase in algorithm $\mathcal{A}^P_{x,y}$. Each triangle corresponds to a light component in $\mathcal{C}_x, \mathcal{C}_y$. The square boxes correspond to the heavy components H_x, H_y. The framed triangles correspond the subtrees of x, y that belong to the set $\mathsf{LDS}(x,y) \cup \mathsf{LDS}(y,x)$. The dashed green bidirectional arrow represents the xy channel given by the promise. The dashed black arrows correspond to the outgoing edges obtained by x, y from the sketch information of their components. In the example, the light subtrees T_{x_2} and T_{y_1} exchange information over their outgoing edge, which allows y to compute the sketch of the merged component $V(T_{x_2}) \cup V(T_{y_1})$. The sketch of the merged component $V(T_{y_h}) \cup V(T_{x_1}) \cup V(T_{x_h})$ is computed by y by letting x send $\mathsf{Sketch}_{G \setminus \{x,y\}}(V(T_{x_1})) \oplus \mathsf{Sketch}_{G \setminus \{x,y\}}(V(T_{x_h}))$.

Õptimal Dual Vertex Failure Connectivity Labels

Merav Parter ✉
Weizmann Institute, Rehovot, Israel

Asaf Petruschka ✉
Weizmann Institute, Rehovot, Israel

Abstract

In this paper we present succinct labeling schemes for supporting connectivity queries under vertex faults. For a given n-vertex graph G, an f-VFT (resp., EFT) connectivity labeling scheme is a distributed data structure that assigns each of the graph edges and vertices a short label, such that given the labels of a vertex pair u and v, and the labels of at most f failing *vertices* (resp., edges) F, one can determine if u and v are connected in $G \setminus F$. The primary complexity measure is the length of the individual labels. Since their introduction by [Courcelle, Twigg, STACS '07], FT labeling schemes have been devised only for a limited collection of graph families. A recent work [Dory and Parter, PODC 2021] provided EFT labeling schemes for general graphs under *edge* failures, leaving the vertex failure case fairly open.

We provide the first sublinear f-VFT labeling schemes for $f \geq 2$ for any n-vertex graph. Our key result is 2-VFT connectivity labels with $O(\log^3 n)$ bits. Our constructions are based on analyzing the structure of dual failure replacement paths on top of the well-known heavy-light tree decomposition technique of [Sleator and Tarjan, STOC 1981]. We also provide f-VFT labels with sub-linear length (in $|V|$) for any $f = o(\log \log n)$, that are based on a reduction to the existing EFT labels.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Fault-Tolerance, Heavy-Light Decomposition, Labeling Schemes

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.32

Related Version *Full Version*: https://arxiv.org/abs/2208.10168

Funding *Merav Parter*: Supported by the European Research Council (ERC) No. 949083), and by the Israeli Science Foundation (ISF) No. 2084/18.

Acknowledgements We would like to thank Michal Dory for useful discussions.

1 Introduction

Connectivity labels are among the most fundamental distributed data-structures, with a wide range of applications to graph algorithms, distributed computing and communication networks. The error-prone nature of modern day communication networks poses a demand to support a variety of logical structures and services, in the presence of vertex and edge failures. In this paper we study fault-tolerant (FT) connectivity labeling schemes, also known in the literature as *forbidden-set* labeling. In this setting, it is required to assign each of the graph's vertices (and possibly also edges) a short name (denoted as *label*), such that given the labels of a vertex pair u and v, and the labels of a faulty-set F, it possible to deduce – using no other information – whether u and v are connected in $G \setminus F$. Since their introduction by Courcelle and Twigg [9] and despite much activity revolving these topics, up until recently FT-labels have been devised only for a restricted collection of graph families. This includes graphs with bounded tree-width, planar graphs, and graphs with bounded doubling dimension [9, 1, 2]. Hereafter, FT-labeling schemes under f faults of vertices (resp., edge) are denoted by f-VFT labeling (resp., f-EFT).

© Merav Parter and Asaf Petruschka;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 32; pp. 32:1–32:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

A recent work by Dory and Parter [11] provided the first EFT-labeling schemes for general n-vertex graphs, achieving poly-logarithmic label length, independent of the number of faults f. For graphs with maximum degree Δ, their labels immediately provide VFT-labels with $\widetilde{O}(\Delta)$ bits[1]. The dependency on Δ is clearly undesirable, as it might be linear in n. This dependency can be explained by the fact that a removal of single vertex might decompose the graph into $\Theta(\Delta)$ disconnected components. The latter behavior poses a challenge for the labeling algorithm that must somehow compress the information on this large number of components into a short label.

While the Δ dependency seems to be inherent in the context of distributed vertex connectivity [35, 31], Baswana and Khanna and Baswana et al. [27, 3] overcome this barrier for the single vertex fault case. Specifically, they presented a construction of distance oracles and labels, that maintain approximate distances in the presence of a single vertex fault with near linear space. This provides, in particular, 1-VFT approximate-distance labels of polylogarithmic length. Their constructions are based on exploiting the convenient structure of single-fault replacement paths. 1-VFT connectivity labels of logarithmic length are easy to achieve using block-cut trees [37], as discussed later on.

When turning to handling dual vertex failures, it has been noted widely that there is a sharp qualitative difference between a single failure and two or more failures. This one-to-two jump has been established by now for a wide-variety of fault-tolerant settings, e.g., reachability oracles [8], distance oracles [12], distance preservers [30, 21, 32] and vertex-cuts [23, 4, 5, 17]. In the lack of any f-VFT labeling scheme with sublinear length for any $f \geq 2$, we focus on the following natural question:

Is it possible to design dual vertex failure connectivity labels of $\widetilde{O}(1)$ length?

The only prior 2-VFT labeling schemes known in the literature have been provided for *directed* graphs in the special case of *single-source* reachabilty by Choudhary [8]. By using the well-known tool of independent trees [18], [8] presented a construction of dual-failure *single-source* reachability data structures, that also provide labels of $O(\log^3 n)$ bits. Note that in a sharp contrast to undirected connectivity that admit $O(\log n)$-bit labels, (all-pairs) reachability labels require *linear* length, even in the fault-free setting [15].

Representation of Small Vertex Cuts: Block-Cut and SPQR Trees. The block-cut tree representation of a graph compactly encodes all of its single cut vertices (a.k.a. articulation points), and the remaining connected components upon the failure of each such vertex [37]. By associating each vertex of the original graph with a corresponding node in the block-cut tree and using standard tree labels techniques, 1-VFT connectivity labels are easily achieved.

Moving on to dual failures, we have the similar (but more complex) SPQR-tree representation [4], which encodes all cut pairs (i.e., vertex pair whose joint failure disconnects the graph). However, it is currently unclear to us how to utilize this structure for 2-VFT connectivity labels. The main issue is generalizing the vertex-node association from the block-cut tree to SPQR tree: each vertex may appear in many nodes with different "virtual edges" adjacent to it, corresponding to different cut-mates forming a cut-pair with it. Kanevsky, Tamassia, Di Battista, and Chen [24] extended the SPQR structure to represent 3-vertex cuts. While these representations are currently limited to cuts of size at most 3, we hope that the approach taken in this paper can be extended to handle larger number of faults. In addition, it is arguably more distributed friendly, as it is based on basic primitives such as the heavy-light tree decomposition, which are easily implemented in the distributed setting.

[1] By including in the label of vertex v the EFT labels of all edges incident to v.

On the Gap Between Edge vs. Vertex Connectivity. Recent years have witnessed an enormous progress in our understanding of vertex cuts, from a pure graph theoretic perspective [34] to many algorithmic applications [29, 28, 34, 22]. Despite this exciting movement, our algorithmic toolkit for handling vertex cuts, especially in the distributed setting, is still considerably limited compared to the counterpart setting of edge connectivity. Near-linear time sequential algorithms for edge connectivity and minimum weighted edge cuts are known for years since the celebrated result of Karger [26], and its recent improvements by [19, 16]. In contrast, only very recently, an almost-linear time algorithm for vertex connectivity has been provided by combining the breakthrough max-flow result of Chen et al. [6] with the work of Li et al. [28]. Turning to the distributed setting, near-optimal congest algorithms for weighted edge-connectivity[2] have been recently presented by Dory et al. [10] and Ghaffari and Zuzic [20]. To this date, there are no distributed algorithms for vertex-connectivity that runs in sublinear number of rounds, for the entire connectivity regime.

Additional Related Work. Our dual-failure vertex connectivity labels are also closely related to *connectivity sensitivity oracles* [13, 14], that provide low-space centralized data-structure for supporting connectivity queries in presence of vertex faults. The main goal in our setting is to provide a *distributed* variant of such construction, where each vertex holds only $S(n)/n$ bits of information, where $S(n)$ is the global space of the centralized data-structure. Duan and Pettie [13, 14] provided an ingenues construction that supports multiple vertex faults in nearly optimal space of $\widetilde{O}(n)$. These constructions are built upon highly centralized building blocks, and their distributed implementation is fairly open.

1.1 Our Contribution

We first present new constructions of 1-VFT and 2-VFT labeling schemes with polylogarithmic length. On a high level, our approach is based on analyzing the structure of dual failure replacement paths, and more specifically their intersection with a given heavy-light tree decomposition of (a spanning tree of) the graph. Throughout, we denote the number of graph vertices (edges) by n (resp., m).

Warm-Up: 1-VFT Connectivity Labels. As a warm-up to our approach, we consider the single fault setting and provide a simple label description that uses only the heavy-light decomposition technique.

▶ **Theorem 1** (1-VFT Connectivity Labels). *For any n-vertex graph, there is a* deterministic *1-VFT connectivity labeling scheme with label length of $O(\log^2 n)$ bits. The decoding algorithm takes* poly$(\log n)$ *time. The labels are computed in $\widetilde{O}(m)$ randomized centralized time, or $\widetilde{O}(D)$ randomized congest rounds.*

While this construction is presented mainly to introduce our technique, it also admits an efficient distributed implementation which follows by the recent work of [33], which we present in detail in the full version.

2-VFT Connectivity Labels. We then turn to consider the considerably more involved setting of supporting two vertex failures. In the literature, heavy-light tree decomposition have been proven useful mainly for handling single vertex faults, e.g. in [27]. The only

[2] In the latter setting, the edges are weighted and it is required to compute the minimum weighted set of edges that disconnects the graph.

dual-failure scheme of [8] is tailored to the *single-source* setting. By carefully analyzing dual-failure replacement paths and their interaction with the heavy-light tree decomposition of a given spanning tree, we provide deterministic labeling schemes of $O(\log^3 n)$ bits. Our main technical contribution in this paper is stated as follow:

▶ **Theorem 2** (2-VFT Connectivity Labels). *For any n-vertex graph, there is a deterministic 2-VFT connectivity labeling scheme with label length of $O(\log^3 n)$ bits. The decoding algorithm takes* $\mathrm{poly}(\log n)$ *time. The labels are computed in $\widetilde{O}(n^2)$ time.*

Our dual-failure labeling scheme uses, in a complete black-box manner, single-source labels. For this purpose, we can use the $O(\log^3 n)$-bit labels of Choudhary [8]. We also provide an alternative construction that is based on the undirected tools of heavy-path tree decomposition, rather than using the tool of independent trees as in [8]. Our single-source labels provide a somewhat improved length of $O(\log^2 n)$ bits[3], but more importantly convey intuition for our all-pairs constructions. Since our labels are built upon a single arbitrary spanning tree that can be assumed to have depth $O(D)$, we are hopeful that this approach is also more distributed-friendly. Specifically, as the depth of the independent trees using in [8] might be linear in n, their distributed computation might be too costly for the purpose of dual vertex cut computation. Moreover, currently the tool of independent trees is limited to only two cut vertices, which also poses a barrier for extending this technique to handle multiple faults. In the full version, we show:

▶ **Lemma 3.** *There is a* single-source *2-VFT connectivity labeling scheme with label length $O(\log^2 n)$ bits. That is, given an n-vertex graph G with a fixed source vertex s, one can label the vertices of G such that given query of vertices $\langle t, x, y\rangle$ along with their labels, the connectivity of s and t in $G \setminus \{x, y\}$ can be inferred.*

f-VFT Connectivity Labels. Finally, we turn to consider labeling schemes in the presence of multiple vertex faults. By combining the notions of sparse vertex certificates [7] with the EFT-labeling scheme of [11], in Appendix A we show:

▶ **Theorem 4** (f-VFT Connectivity Labels). *There is a f-VFT connectivity labeling scheme with label length $\widetilde{O}(n^{1-1/2^{f-2}})$ bits, hence of sublinear length of any $f = o(\log\log n)$.*

This for example, provides 3-VFT labels of $\widetilde{O}(\sqrt{n})$ bits.

1.2 Preliminaries

Given a connected n-vertex graph $G = (V, E)$, we fix an arbitrary source vertex $s \in V$, and a spanning tree T of G rooted at s. We assume each vertex a is given a unique $O(\log n)$-bit identifier $\mathrm{ID}(a)$. Let $par(a)$ be the parent of a in T, T_a be the subtree of T rooted at a, and T_a^+ be the tree obtained from T_a by connecting $par(a)$ to a. The (unique) tree path between two vertices a, b is denoted $T[a, b]$.[4] Let $\mathrm{depth}(a)$ be the hop-distance of vertex a from the root s in T, i.e. the number of edges $T[s, a]$. We say that vertex a is *above* or *higher* (resp., *below* or *lower*) than vertex b if $\mathrm{depth}(a)$ is smaller (resp., larger) than $\mathrm{depth}(b)$. The vertices a, b are said to be *dependent* if one of them is an ancestor of the other in T, and *independent* otherwise.

[3] We note that it might also be plausible to improve the label size of [8] to $O(\log^2 n)$ bits, by reducing the size of their range-minima labels.

[4] Note that $T[a, b]$ is a path, but T_a is a subtree.

For two paths $P, Q \subseteq G$, define the concatenation $P \circ Q$ as the path formed by concatenating Q to the end of P. The concatenation is well defined if for the last vertex p_ℓ of P and the first vertex q_f of Q it either holds that $p_\ell = q_f$ or that $(p_\ell, q_f) \in E$. We use the notation $P(a, b]$ for the subpath of P between vertices a and b, excluding a and including b. The subpaths $P[a, b)$, $P[a, b]$ and $P(a, b)$ are defined analogously. We extend this notation for tree paths, e.g. $T(a, b]$ denotes the subpath of $T[a, b]$ obtained by omitting a. When we specify P as an a-b path, we usually think of P as directed from a to b. E.g., a vertex $c \in P$ is said to be the *first* having a certain property if it is the closest vertex to a among all vertices of P with the property. A path P *avoids* a subgraph $H \subseteq G$ if they are vertex disjoint, i.e. $V(P) \cap V(H) = \emptyset$.

For a subgraph $G' \subseteq G$, let $\deg(a, G')$ be the degree of vertex a in G'. We denote by $\mathsf{conn}(a, b, G')$ the connectivity status of vertices a and b in G', which is 1 if a and b are connected in G' and 0 otherwise. We give arbitrary unique $O(\log n)$-bit IDs to the connected components of G', e.g. by taking the maximal vertex ID in each component. We denote by $\mathsf{CID}(a, G')$ the ID of the connected component containing vertex a in G'. For a failure (or fault) set $F \subseteq V$, we say that two vertices a, b are *F-connected* if $\mathsf{conn}(a, b, G \setminus F) = 1$, and *$F$-disconnected* otherwise. In the special cases where $F = \{x\}$ or $F = \{x, y\}$ for some $x, y \in V$, we use respectively the terms x-connected or xy-connected.

Replacement Paths. For a given (possibly weighted) graph G, vertices $a, b \in V$ and a faulty set $F \subseteq V$, the *replacement path* $P_{a,b,F}$ is the shortest a-b path in $G \setminus F$. In our context, as we are concerned with connectivity rather than in shortest-path distances, we assign weights to the graph edges for the purpose of computing replacement paths with some convenient structure w.r.t a given spanning tree T. Specifically, by assigning weight of 1 to the T-edges, and weight n to non T-edges, the resulting replacement paths "walk on T whenever possible". Formally, this choice of weights ensures the following property of the replacement paths: for any two vertices $c, d \in P_{a,b,F}$ such that $T[c, d] \cap F = \emptyset$, $P_{a,b,F}[c, d] = T[c, d]$. Also note that these replacement paths are shortest w.r.t our weight assignment, but might not be shortest w.r.t their number of edges. We may write $P_{a,b,x}$ when $F = \{x\}$.

Heavy-Light Tree Decomposition. Our labeling schemes use the classic heavy-light tree decomposition technique introduced by Sleator and Tarjan [36]. This is inspired by the work of Baswana and Khanna [27] applying this technique in the fault-tolerant setting. The *heavy child* of a non-leaf vertex a in T, denoted $h(a)$, is the child b of a that maximizes the number of vertices in its subtree T_b (ties are broken arbitrarily in a consistent manner.). A vertex is called *heavy* if it is the heavy child of its parent, and *light* otherwise. A tree edge in T is called *heavy* if it connects a vertex to its heavy child, and *light* otherwise. The set of heavy edges induces a collection of tree paths, which we call *heavy paths*. Let $a, b \in V$ such that a is a strict ancestor of b, and let a' be the child of a on $T[a, b]$. Then a is called a *heavy ancestor* of b if a' is heavy, or a *light ancestor* of b if a' is light. Note that a heavy ancestor of b need not be a heavy vertex itself, and similarly for light ancestors. We observe that if b is a light child of a, then T_b contains at most half of the vertices in T_a. Consequently, we have:

▶ **Observation 5.** *Any root-to-leaf path in T contains only $O(\log n)$ light vertices and edges.*

Our labeling schemes are based on identifying for each vertex a a small number of *interesting* vertices, selected based on the heavy-light decomposition.

▶ **Definition 6.** *The* interesting set *of a vertex a is defined to be $I(a) = \{b \in T[s,a] \mid b$ is light$\} \cup \{h(a)\}$ (where $\{h(a)\}$ is interpreted as the empty set if a is a leaf). That is, $I(a)$ consists of all the light vertices on $T[s,a]$, along with the heavy child of a (if it exists). The* upper-interesting set *of a is defined to be $I^{\uparrow}(a) = \{par(b) \mid b \in I(a)\} \cup \{a\}$. That is, $I^{\uparrow}(a)$ consists of all the light ancestors of a and a itself.*

We make extensive use of the following useful properties of interesting sets, which are immediate to prove.

▶ **Lemma 7.** *For any $a \in V$, $|I(a)| = O(\log n)$ and $|I^{\uparrow}(a)| = O(\log n)$.*

▶ **Lemma 8.** *Let $a, b \in V$ such that $a \in T[s,b]$.*
(1) *If $a \neq b$, then for the child a' of a on $T[a,b]$ it holds that $a' \in I(a) \cup I(b)$.*
(2) *If $a \notin I^{\uparrow}(b)$, then $a \neq b$ and the child of a on $T[a,b]$ is $h(a)$.*

Extended Vertex IDs. To avoid cumbersome definitions in our labels, it is convenient to augment the vertex IDs with additional $O(\log n)$ bits of information, resulting in *extended IDs*. The main ingredient is *ancestry labels* [25]: these are $O(\log n)$-bit labels $\mathsf{ANC}_T(a)$ for each vertex a, such that given $\mathsf{ANC}_T(a)$ and $\mathsf{ANC}_T(b)$ one can infer whether a is an ancestor of b in T. The extended ID of a vertex a is[5] $\mathrm{EID}(a) = [\mathrm{ID}(a), \mathsf{ANC}_T(a), \mathrm{ID}(h(a)), \mathsf{ANC}_T(h(a))]$. Thus, given $\mathrm{EID}(a)$ and $\mathrm{EID}(b)$, one can determine whether a is an ancestor of b in T, and moreover, whether it is a light or a heavy ancestor. We will not explicitly refer to the extended IDs, but rather use them as follows:
- The label of any vertex a always (implicitly) stores $\mathrm{EID}(a)$.
- Whenever a label stores a given vertex a, it additionally stores $\mathrm{EID}(a)$.

This enables us to assume throughout that we can always determine the (heavy or light) ancestry relations of the vertices at play.

2 Single Failure Connectivity Labels

In this section we warm-up by considering the single failure case of Theorem 1.

The construction of the labels for each vertex a is described in Algorithm 1. The label length of $O(\log^2 n)$ bits follows by Lemma 7.

Algorithm 1 Construction of label $\mathsf{L}_{\mathsf{1F}}(a)$ for vertex a.

```
1 for each b' ∈ I(a) with par(b') = b do
2     store vertices b, b' and the values conn(s, b', G \ {b}), CID(b', G \ {b});
```

The key observation for decoding is:

▷ Claim 9. Given $\mathsf{L}_{\mathsf{1F}}(w)$ and $\mathsf{L}_{\mathsf{1F}}(x)$, one can determine the x-connectivity of w and s, and also find $\mathsf{CID}(w, G \setminus \{x\})$ in case w, s are x-disconnected.

Proof. If w is not a descendant[6] of x, then $T[s,w]$ is failure-free, so w and s are x-connected and we are done. Assume now that w is a descendant of x, and let x' be the child of x on $T[x,w]$. Then $T[x',w]$ is failure-free, hence x' and w are x-connected. Therefore, it suffices

[5] If a is a leaf, we simply omit from $\mathrm{EID}(a)$ the information regarding $h(a)$.
[6] This is checked using extended IDs $\mathrm{EID}(w)$ and $\mathrm{EID}(x)$.

to determine the values $\mathsf{conn}(s, x', G \setminus \{x\})$ and $\mathsf{CID}(x', G \setminus \{x\})$. Lemma 8(1) guarantees that $x' \in I(w) \cup I(x)$, hence the required values are stored either in $\mathsf{L}_{\mathsf{1F}}(w)$ or in $\mathsf{L}_{\mathsf{1F}}(x)$ (by setting $b = x$ and $b' = x'$). $\lhd$

Given $\mathsf{L}_{\mathsf{1F}}(u)$, $\mathsf{L}_{\mathsf{1F}}(v)$ and $\mathsf{L}_{\mathsf{1F}}(x)$, we determine the x-connectivity of u, v as follows. We apply Claim 9 twice, with $w = u$ and with $w = v$. If we find the component IDs of both u and v in $G \setminus \{x\}$, we compare them and answer accordingly. However, if this is not the case, then we must discover that one of u, v is x-connected to s, so we should answer affirmatively iff the other is x-connected to s. This completes the decoding algorithm of Theorem 1. The preprocessing time analysis appears in the full version.

3 Dual Failure Connectivity Labels

3.1 Technical Overview

In the following we provide high-level intuition for our main technical contribution of dual failure connectivity labels. Throughout, the query is given by the tuple $\langle u, v, x, y\rangle$, where x, y are the vertex faults. Recall that our construction is based on some underlying spanning tree T rooted at some vertex s (that we treat as the *source*). Similarly to the 1-VFT construction, we compute the heavy-light tree decomposition of T, which classifies the tree edges into heavy and light.

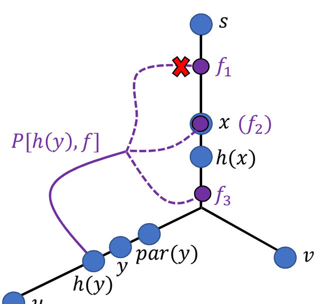
We distinguish between two structural cases depending on the locations of the two faults, x and y. The first case which we call *dependent* handles the setting where x and y have ancestry/descendant relations. The second *independent* case assumes that x and y are not dependent, i.e., $\mathrm{LCA}(x, y) \notin \{x, y\}$, where $\mathrm{LCA}(x, y)$ is the lowest (or least) common ancestor of x and y in T.

Our starting observation is that by using single-source 2-VFT labels in a black-box manner, we may restrict our attention to the hard case where the source s is xy-disconnected from both u and v. Quite surprisingly, this assumption yields meaningful restrictions on the structure of key configurations, as will be demonstrated shortly.

Dependent Failures. To gain intuition, we delve into two extremes: the easy *all-light* case where u, v, y are all *light* descendants of x, and the difficult *all-heavy* case where they are all *heavy* descendants of x. Consider first the easy all-light case. As every vertex has only $O(\log n)$ light ancestors, each of u, v, y has the budget to prepare by storing its 1-VFT label w.r.t the graph $G \setminus \{x\}$. Then, for decoding, we simply answer the single-failure query $\langle u, v, y\rangle$ in $G \setminus \{x\}$.

We turn to consider the all-heavy case, which turns out to be an important core configuration. Here, we no longer have the budget to prepare for each possible failing x, and a more careful inspection is required. The interesting case is when $y \in T[u, v]$. We further focus in this overview on the following instructive situation: y is not an ancestor of v, but is a *heavy* ancestor of u. It is then sufficient to determine the xy-connectivity of $h(y)$ and $par(y)$. See Figure 1 (left). Naturally, y is most suited to prepare in advance for this situation, as follows. Let $P = P_{h(y), par(y), y}$ be the $h(y)$-$par(y)$ replacement path avoiding y, and let $f \in P$ be the *first* vertex (i.e., closest to $h(y)$) from $T[s, par(y)]$. Surprisingly, it suffices for the labeling algorithm to include in the label of y the identity of f, along with a single bit representing the connectivity of $h(y)$ and $par(y)$ in $G \setminus \{f, y\}$.

This limited amount of information turns out to be sufficient thanks to the useful structures of the replacement paths. Since x is an ancestor of y, we need to consider the possible locations of x within $T[s, par(y)]$. The key observation is that x cannot lie below



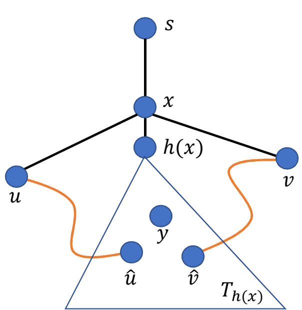


Figure 1 Left: Illustration of the all-heavy configuration. Letting $P = P_{h(y),par(y),y}$, the purple path represents the prefix $P[h(y), f]$ of P until the first time it hits $T[s, par(y)]$. Vertices f_1, f_2, f_3 correspond to different options for the location of f: above x, equals x, or below x. The f_1 option is marked X as it is excluded by our analysis. Right: Illustration of the reduction to the all-heavy case. The analog vertices $\widehat{u}, \widehat{v}$ are chosen from $\mathsf{AnSet}(u, x), \mathsf{AnSet}(v, x)$, respectively.

f, i.e. in $T(f, par(u)]$: otherwise, $T[u, h(y)] \circ P[h(y), f] \circ T[f, s]$ is a u-s path avoiding x, y, which we assume does not exist! Now, if x is above f, i.e. in $T[s, f)$, then P is fault-free, so we determine that $h(y), par(y)$ are xy-connected. If $x = f$, we simply have the answer stored explicitly by the label of y. The complete solution for the all-heavy case is of a similar flavor, albeit somewhat more involved.

We then handle the general dependent failures case by reducing to the all-heavy configuration, which we next describe in broad strokes. First, the case where y is a light descendant of x is handled directly using 1-VFT labels, in a similar manner to the all-light case. In the remaining case where $y \in T_{h(x)}$, a challenge arises when (at least) one of u, v, say u, is a light descendant of x. We exploit the fact that the label of u has the *budget* to prepare for light ancestors, and store in this label a small and carefully chosen set of vertices from $T_{h(x)}$, called the *analog set* $\mathsf{AnSet}(u, x)$. Our decoding algorithm in this case replaces the given $\langle u, v, x, y\rangle$ query with an *analogous* all-heavy query $\langle \widehat{u}, \widehat{v}, x, y\rangle$ for some $\widehat{u} \in \mathsf{AnSet}(u, x)$ and $\widehat{v} \in \mathsf{AnSet}(v, x)$. See Figure 1 (right). The reduction's correctness is guaranteed by the definition of analog sets.

Independent Failures. Our intuition comes from our solution to the *single-source* independent-failures case, described in the full version. As we assume that both u, v are xy-disconnected from s, we know that the corresponding decoding algorithm *rejects* both queries $\langle u, x, y\rangle$ and $\langle v, x, y\rangle$. Rejection instances can be of two types: *explicit reject* or *implicit reject.*

If $\langle u, x, y\rangle$ is an explicit reject instance, then the algorithm rejects by tracking down an explicit bit stored in one of the labels of u, x, y, and returning it. This bit is of the form $\mathsf{conn}(s, \widetilde{u}, G \setminus \{x, y\})$ for some vertex $\widetilde{u}$ which is xy-connected to u. So, in explicit reject instances, one of the vertices u, x, y has *prepared in advance* by storing this bit. In contrast, if it is an implicit reject instance, then the algorithm detects *at query time* that $\langle u, x, y\rangle$ match a specific, highly-structured fatal configuration leading to rejection. Specifically, this configuration implies that u is xy-connected to both $h(x)$ and $h(y)$. Thus, when reaching implicit rejection, we can infer useful structural information.

Getting back to our original query $\langle u, v, x, y\rangle$, the idea is to handle all of the four possible combinations of implicit or explicit rejects for $\langle u, x, y\rangle$ or $\langle v, x, y\rangle$. Our approach is then based on augmenting the single-source 2-VFT labels in order to provide the decoding algorithm with a richer information in the explicit rejection cases.

The presented formal solution distills the relevant properties of the corresponding (augmented) single-source labels and decoding algorithm. This approach has the advantage of having a succinct, clear and stand-alone presentation which does not require any prior knowledge of our single-source solution, but might hide some of the aforementioned intuition.

Setting Up the Basic Assumptions. We now precisely describe the basic assumptions that we enforce as preliminary step. These are:

(C1) u and v are both x-connected and y-connected.

(C2) Both u and v are xy-disconnected from the source s.

To verify condition (C1), we augment the 2-VFT label of each vertex with its 1-VFT label from Theorem 1. If (C1) is not satisfied, then clearly u, v are xy-disconnected, so we are done. For (C2), we further augment the labels with the corresponding single-source 2-VFT labels of [8] (or alternatively, our own such labels of Lemma 3). Using them we check whether u and v are xy-connected to s. If both answers are affirmative, then u, v are xy-connected. If the answers are different, then u, v are xy-disconnected. Hence, the only non-trivial situation is when (C2) holds. As the 1-VFT and single-source 2-VFT labels consume only $O(\log^3 n)$ bits each, the above mentioned augmentations are within our budget.

The following sections present our 2-VFT connectivity labeling scheme in detail: Section 3.2 handles the independent-failures case, and Section 3.3 considers the dependent-failure case. The final label is obtained by adding the sublables provided in each of these sections. We note that by using the extended IDs of the vertices, it is easy for the decoding algorithm to detect which of the cases fits the given $\langle u, v, x, y\rangle$ query.

3.2 Two Failures are Independent

The independence of the failures allows us to enforce a stronger version of condition (C1):

(C3) u, v and s are all x-connected and y-connected.

Condition (C3) is verified using the 1-VFT labels of u, v, x, y and s. As the label of s is not given to us, we just augment the label of every vertex also with the 1-VFT label of s. If (C3) fails, we are done by the following claim.

▷ Claim 10. If condition (C3) does not hold, then u and v are xy-connected.

Proof. Assume (C3) does not hold. By (C1), this can happen only if one of u, v, say u, is disconnected from s under one of the failures, say x. Namely, u, s are x-disconnected. Now, (C1) also ensures that there is a u-v path P avoiding x. We assert that P also avoids y, which completes the proof. Assume otherwise, and consider the u-s path $P' = P[u, y] \circ T[y, s]$. By the independence of x, y we have that $x \notin P'$, which contradicts the fact that u, s are x-disconnected. ◁

Our general strategy is to design labels $\mathsf{L_P}(a)$ for each vertex a that have following property:

(P) For any $\langle u, v, x, y\rangle$ with independent failures[7] x, y, there exists[8] $z \in \{h(x), h(y)\}$ such that given the label $\mathsf{L_P}(w)$ of any $w \in \{u, v\}$ and the labels $\mathsf{L_P}(x), \mathsf{L_P}(y)$, one can infer the xy-connectivity of w, z, and also find $\mathsf{CID}(w, G \setminus \{x, y\})$ in case w, z are xy-disconnected.

This suffices to determine the xy-connectivity of u, v by the following lemma:

▶ **Lemma 11.** *Given the $\mathsf{L_P}$ labels of u, v, x, y, one can determine the xy-connectivity of u, v.*

[7] Which satisfy conditions (C1), (C2) and (C3).

[8] At least one of $h(x), h(y)$ exists: else, x, y are leaves, so $T \setminus \{x, y\}$ spans $G \setminus \{x, y\}$, contradicting (C2).

Proof. We apply property (P) twice, for $w = u$ and for $w = v$. If we find the component IDs of both u and v in $G \setminus \{x, y\}$ we just compare them and answer accordingly. However, if this does not happen, then we must discover that one of u, v is xy-connected to z, so we should answer affirmatively iff the other is also xy-connected to z. ◀

In order to preserve the symmetry between the independent failures x and y (which we prefer to break in a more favorable manner in our subsequent technical arguments), we do not explicitly specify, at this point, the identity of $z \in \{h(x), h(y)\}$. It may be useful for the reader to think of z as chosen *adversarially* from $\{h(x), h(y)\}$, and our decoding algorithm handles each of the two possible selections. Alternatively, this can be put as follows: our labeling scheme will guarantee property (P) for $z = h(x)$ *and* for $z = h(y)$ (if both heavy children exist).

Construction of $\mathsf{L_P}$ Labels. We start with a useful property of single-fault replacement paths. For a vertex $a \in V$ with an s-a replacement path $P = P_{s,a,par(a)}$, let $\ell_a \in P$ be the last (closest to a) vertex in $T \setminus T_{par(a)}$.

▶ **Observation 12.** $P_{s,a,par(a)} = T[s, \ell_a] \circ Q$ *where* $Q \subseteq T_{par(a)}$.

We are now ready to define the $\mathsf{L_P}$ labels. These are constructed by Algorithm 2. The label length of $O(\log^3 n)$ bits follows by Lemma 7.

Algorithm 2 Construction of label $\mathsf{L_P}(a)$ for vertex a.

```
1 for each b′ ∈ I(a) with par(b′) = b do
2     store vertices b, b′, ℓ_{b′};
3     for each c ∈ I↑(ℓ_{b′}) do
4         store vertex c;
5         store CID(b′, G \ {b, c}), conn(b′, h(b), G \ {b, c}), conn(b′, h(c), G \ {b, c});
```

Decoding Algorithm for Property (P). Our goal is to show that given $\mathsf{L_P}(w)$ for $w \in \{u, v\}$ and $\mathsf{L_P}(x), \mathsf{L_P}(y)$ we can indeed satisfy the promise of (P); namely, determine the xy-connectivity of w, z, and in case they are xy-disconnected also report $\mathsf{CID}(w, G \setminus \{x, y\})$.

One of the failures, say x, must be an ancestor of w in T. Otherwise, w would have been connected to s in $G \setminus \{x, y\}$, contradicting (C2). Denote by x' the child of x on $T[x, w]$. The independence of x, y guarantees that $T[x', w]$ is fault-free, hence x', w are xy-connected.

It now follows from (C2) that x', s are xy-disconnected, and from (C3) that x', s are x-connected. The latter ensures that $\ell_{x'}$ is well-defined. By Observation 12, $P_{s,x',x} = T[s, \ell_{x'}] \circ Q$ where $Q \subseteq T_x$, so $y \notin Q$. On the other hand, it cannot be $P_{s,x',x}$ entirely avoids y, as we have already established that s, x' are xy-disconnected. It follows that $y \in T[s, \ell_{x'}]$. See illustration in Figure 2. Note that $x' \in I(w) \cup I(x)$ by Lemma 8, hence the triplet of vertices $b = x$, $b' = x'$ and $\ell_{b'} = \ell_{x'}$ is stored either in $\mathsf{L_P}(w)$ or in $\mathsf{L_P}(x)$. We next distinguish between two cases, depending on whether y belongs to the upper-interesting set of $\ell_{x'}$.

Case 1: $y \in I^{\uparrow}(\ell_{x'})$. Then the following are also specified in the last label (with $c = y$):

$$\mathsf{CID}(x', G \setminus \{x, y\}),\ \mathsf{conn}(x', h(x), G \setminus \{x, y\}),\ \mathsf{conn}(x', h(y), G \setminus \{x, y\}).$$

Since x', w are xy-connected, we can replace x' by w in the three values above, and reporting them guarantees property (P).

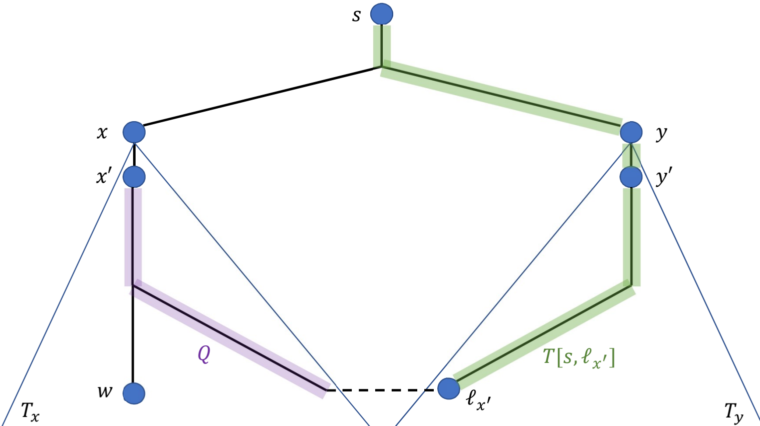


Figure 2 Illustration of the decoding algorithm for the independent failures case. The path $T[s, \ell_{x'}]$ is shown in green (right), and the path Q is shown in purple (left). The concatenation $T[s, \ell_{x'}] \circ Q$ forms the replacement path $P_{s,x',x}$. The vertex y' is the child of y on $T[s, \ell_{x'}]$. The case where $y \notin I^{\uparrow}(\ell_{x'})$ corresponds to $y' = h(y)$.

Case 2: $y \notin I^{\uparrow}$ $(\ell_{x'})$**.** Then the child of y on $T[y, \ell_{x'}]$ is $h(y)$ by Lemma 8. Thus $T[h(y), \ell_{x'}] \circ Q$ is a $h(y)$-x' path avoiding both x and y. Therefore, as w is xy-connected to x', it is also xy-connected to $h(y)$. So, if $z = h(y)$ we are done. However, if $z = h(x)$, we recover by simply repeating the algorithm when $y, h(y), x$ play the respective roles of x, x', y. The triplet $y, h(y), \ell_{h(y)}$ is stored $\mathsf{L_P}(y)$ since $h(y) \in I(y)$. If $x \in I^{\uparrow}(\ell_{h(y)})$, this label also specifies

$$\mathsf{CID}(h(y), G \setminus \{x, y\}),\ \mathsf{conn}(h(y), h(x), G \setminus \{x, y\}),$$

and since $w, h(y)$ are xy-connected we can replace $h(y)$ by w in these values, and report them to guarantee property (P). Otherwise, we deduce (by a symmetric argument) that w and $h(x) = z$ are xy-connected, so we are done. This concludes the decoding algorithm for property (P). Finally, by Lemma 11, we obtain:

▶ **Lemma 13.** *There are $O(\log^3 n)$-bit labels $\mathsf{L_{ind}}$ supporting the independent failures case.*

3.3 Two Failures are Dependent

In this section, we consider the complementary case where x and y are dependent. As previously discussed, our strategy is based on reducing the general dependent-failures case to the well-structured configuration of the all-heavy case:

▶ **Definition 14.** *A query of vertices $\langle u, v, x, y\rangle$ is said to be* all-heavy (AH) *if $u, v, y \in T_{h(x)}$.*

We first handle this configuration in Section 3.3.1 by defining sub-labels $\mathsf{L_{AH}}$ that are tailored to handle it. Then, Section 3.3.2 considers the general dependent-failures case.

3.3.1 The All-Heavy (AH) Case

Construction of $\mathsf{L_{AH}}$ Labels. We observe another useful property of single-fault replacement paths. For a vertex $a \in V$ with an a-s replacement path $P = P_{a,s,par(a)}$, let $f_a \in P$ be the first (closest to a) vertex in $T[s, par(a))$.

▶ **Observation 15.** *$P_{a,s,par(a)} = Q \circ T[f_a, s]$ where Q avoids $T[s, par(a)]$.*

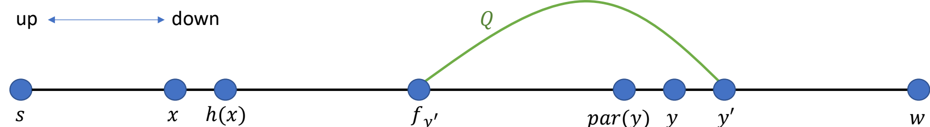


Figure 3 Illustration of the proof of Claim 16. The tree path from s to w is shown sideways, where the depth increases from left to right. The path Q appears in green.

We are now ready to define the $\mathsf{L_{AH}}$ labels. These are constructed by Algorithm 3. The label length of $O(\log^2 n)$ bits follows by Lemma 7.

Algorithm 3 Construction of label $\mathsf{L_{AH}}(a)$ for vertex a.

1 **for** *each* $b' \in I(a)$ *with* $par(b') = b$ **do**
2   **store** vertices $b, b', f_{b'}$;
3   **store** $\mathsf{conn}(b', par(b), G \setminus \{b, f_{b'}\})$, $\mathsf{CID}(b', G \setminus T[s, b])$;

Decoding Algorithm for (AH) Case. Assume we are given an (AH)-query $\langle u, v, x, y\rangle$ along with the $\mathsf{L_{AH}}$ labels of these vertices. The main idea behind the construction of the $\mathsf{L_{AH}}$ labels is to have:

▷ Claim 16. If $w \in \{u, v\}$ is a descendant of y, then given the $\mathsf{L_{AH}}$ labels of w, x, y, one can:
(1) find $\mathsf{CID}(w, G \setminus T[s, y])$, and
(2) determine whether w and $par(y)$ are xy-connected.

Proof. Let y' be the child of y on $T[y, w]$. Then y', w are xy-connected as $T[y', w]$ is fault-free. Hence, in the following we can replace w by y' for determining both (1) and (2). Also, as w, y' are (particularly) y-connected, and w, s are y-connected by (C1), we have that y', s are y-connected, so $f_{y'}$ is well-defined. By Lemma 8, it holds that $y' \in I(w) \cup I(y)$, hence the triplet $b = y$, $b' = y'$ and $f_{b'} = f_{y'}$ is stored either in $\mathsf{L_{AH}}(w)$ or in $\mathsf{L_{AH}}(y)$. The same label also includes $\mathsf{CID}(y', G \setminus T[s, y])$, so (1) follows. We also find there the value $\mathsf{conn}(y', par(y), G \setminus \{y, f_{y'}\})$. Note that if $f_{y'} = x$, then (2) follows as well. Assume now that $f_{y'} \neq x$. By Observation 15, the path $P_{y',s,y}$ is of the form $Q \circ T[f_{y'}, s]$ where Q avoids $T[s, y]$. We now observe that $f_{y'} \notin T[s, x)$: this follows as otherwise, the w-s path given by $T[w, y'] \circ Q \circ T[f_{y'}, s]$ is failure-free, contradicting (C2). As $f_{y'}$ is, by definition, a vertex in $T[s, y)$, it follows that $f_{y'} \in T(x, y)$. The y'-$par(y)$ path $Q \circ T[f_{y'}, par(y)]$ now certifies that $y', par(y)$ are xy-connected, which gives (2). See illustration in Figure 3. ◁

We next show how to use Claim 16 for determining the xy-connectivity of u, v. The proof divides into three cases according to the ancestry relations between u, v and y.

Case 1: Neither of u, v is a descendant of y. Then $y \notin T[u, v]$. Since both $u, v \in T_{h(x)}$, also $x \notin T[u, v]$. Thus u, v are xy-connected, and we are done.

Case 2: Only one of u, v is a descendant of y. W.l.o.g., assume the descendant is u. Then $y \neq h(x)$, as otherwise v would also be a descendant of y. It follows that $par(y) \in T_{h(x)}$. Hence $T[par(y), v] \subseteq T_{h(x)}$, so it avoids x. It also avoids y, as $T[par(y), v]$ contains only ancestors of $par(y)$ or of v. Thus, $v, par(y)$ are xy-connected. Finally, we use Claim 16(2) with $w = u$ to determine the xy-connectivity of $u, par(y)$, or equivalently of u, v.

Case 3: Both u, v are descendants of y. We apply Claim 16 twice, with $w = u$ and $w = v$. Using (2), we check for both u and v if they are xy-connected to $par(y)$. The only situation in which we cannot infer the xy-connectivity of u, v is when both answers are negative. When this happens, we exploit (1) and compare the component IDs of u, v in $G \setminus T[s, y]$. If they are equal, then clearly u, v are xy-connected as $x, y \in T[s, y]$. Otherwise, we assert that we can safely determine that u, v are xy-disconnected.

▷ Claim 17. If (i) both u and v are xy-disconnected from $par(y)$, and (ii) u and v are $T[s, y]$-disconnected, then u, v are xy-disconnected.

Proof. Assume towards a contradiction that there exists a u-v path P in $G \setminus \{x, y\}$. By (ii), P must intersect $T[s, y]$. Let a be a vertex in $P \cap T[s, y]$. As $a \notin \{x, y\}$, it holds that either $a \in T[s, x)$ or $a \in T(x, y)$. If $a \in T[s, x)$, then the path $P[u, a] \circ T[a, s]$ connects u to s in $G \setminus \{x, y\}$, but this contradicts (C2). If $a \in T(x, y)$, then the path $P[u, a] \circ T[a, par(y)]$ connects u to $par(y)$ in $G \setminus \{x, y\}$, contradicting (i). ◁

This concludes the decoding algorithm for the (AH) case, and proves:

▶ **Lemma 18.** *There exist $O(\log^2 n)$-bit labels L_{AH} supporting the all-heavy (AH) case.*

3.3.2 The General Dependent-Failures Case

We assume w.l.o.g. that y is a descendant[9] of x in T. Condition (C2) implies that both u and v are also descendants of x. Recall that our general strategy is reducing to the (AH) case, as follows. First, the case where y is not in $T_{h(x)}$ is handled by using 1-VFT labels *in the graph* $G \setminus \{x\}$, which enable us to determine directly whether u, v are connected in $(G \setminus \{x\}) \setminus \{y\} = G \setminus \{x, y\}$. In the remaining case where $y \in T_{h(x)}$, we show how to "replace" u, v by *xy-analogs*: vertices $\widehat{u}, \widehat{v}$ that are xy-connected to u, v (respectively), and lie inside $T_{h(x)}$. Thus, the query $\langle \widehat{u}, \widehat{v}, x, y \rangle$ is an analogous (AH)-query to answer.

Construction of $\mathsf{L}_{\mathsf{dep}}$ Labels. The construction is based on defining, for any given vertex a and ancestor b of a, a small set of vertices from $T_{h(b)}$, serving as candidates to be bc-analogs of a for *any* $c \in T_{h(b)}$.

▶ **Definition 19** (Analog Sets). *For $a, b \in V$ such that b is an ancestor of a in T, the* analog set $\mathsf{AnSet}(a, b)$ *consists of two (arbitrary) distinct vertices c_1, c_2 with the following property: $c_i \in T_{h(b)}$ and there exists an a-c_i path avoiding $T^+_{h(b)} \setminus \{c_i\}$. If there is only one such vertex, then* $\mathsf{AnSet}(a, b)$ *is the singleton containing it, and if there are none then* $\mathsf{AnSet}(a, b) = \emptyset$.

The $\mathsf{L}_{\mathsf{dep}}$ labels are constructed by Algorithm 4. We use the notation $\mathsf{L}_{\mathsf{1F}}(a, G')$ to denote the 1-VFT label of $a \in V$ from Theorem 1 *constructed w.r.t the subgraph $G' \subseteq G$*. The label length of $O(\log^3 n)$ bits follows by Lemma 7, Theorem 1 and Lemma 18.

Decoding Algorithm for Dependent Failures. Assume we are given a dependent-failures query $\langle u, v, x, y \rangle$ where y is a descendant of x, along with the $\mathsf{L}_{\mathsf{dep}}$ labels. We first treat the easier case where x is a light ancestor of y using the 1-VFT labels in $G \setminus \{x\}$.

[9] We can check this using the extended IDs, and swap the roles of x in y if needed.

Algorithm 4 Construction of label $\mathsf{L_{dep}}(a)$ for vertex a.

```
1 store L_AH(a);
2 for each b' ∈ I(a) with par(b') = b do
3     store vertices b, b';
4     store L_1F(a, G \ {b}), L_1F(b', G \ {b});
5     store vertex set AnSet(a, b), and L_AH(c_i) for each c_i ∈ AnSet(a, b);
6     store CID(a, G \ T^+_{h(b)});
```

Case: x is a light ancestor of y. Then the child x_y of x on the path $T[x, y]$ is light, hence $x_y \in I(y)$. Therefore, the label of y contains the 1-VFT label $\mathsf{L_{1F}}(y, G \setminus \{x\})$. Let x_u be the child of x on the path $T[x, u]$, and define

$$\widetilde{u} = \begin{cases} \text{if } x_u \text{ is light:} & u, \\ \text{if } x_u \text{ is heavy:} & x_u = h(x). \end{cases}$$

▷ Claim 20. It holds that (i) $\widetilde{u}$ is xy-connected to u, and (ii) one can find $\mathsf{L_{1F}}(\widetilde{u}, G \setminus \{x\})$.

Proof. If x_u is light: Then (i) is trivial. For (ii), note that $x_u \in I(u)$, hence the 1-VFT label of $\widetilde{u} = u$ with respect to $G \setminus \{x\}$, which is $\mathsf{L_{1F}}(u, G \setminus \{x\})$, is stored in $\mathsf{L_{dep}}(u)$.

If x_u is heavy: Then $x_u \neq x_y$, hence the path $T[x_u, u]$ is failure-free, which proves (i). For (ii), note that $x_u = h(x) \in I(x)$, hence the 1-VFT label of $\widetilde{u} = h(x)$ with respect to $G \setminus \{x\}$, which is $\mathsf{L_{1F}}(h(x), G \setminus \{x\})$, is stored in $\mathsf{L_{dep}}(x)$. ◁

We define $\widetilde{v}$ and find its 1-VFT label with respect to $G \setminus \{x\}$ in a similar fashion. Finally, we use the 1-VFT labels to answer the *single failure* query $\langle \widetilde{u}, \widetilde{v}, y \rangle$ with respect to the graph $G \setminus \{x\}$, which determines the xy-connectivity of $\widetilde{u}, \widetilde{v}$, or equivalently of u, v. So in this case, the decoding algorithm directly determine the xy-connectivity of u, v, and we are done.

From now on, we assume that x is a heavy ancestor of y, or equivalently that $y \in T_{h(x)}$.

Replacing u, v with their xy-analogs. In the following, we restrict our attention to u (and the same can be applied to v). Again, let x_u be the child of x on $T[s, u]$. The xy-analog $\widehat{u}$ of u is defined as:

$$\widehat{u} = \begin{cases} \text{if } x_u \text{ is heavy:} & u, \\ \text{if } x_u \text{ is light and } \mathsf{AnSet}(u, x) \setminus \{y\} \neq \emptyset\text{:} & c \in \mathsf{AnSet}(u, x) \setminus \{y\}, \\ \text{if } x_u \text{ is light and } \mathsf{AnSet}(u, x) \setminus \{y\} = \emptyset\text{:} & \text{undefined.} \end{cases}$$

The corner case where $\widehat{u}$ is undefined can be alternatively described as follows:

(C4) x_u is light, and no $c \in T_{h(x)} \setminus \{y\}$ is connected to u by a path internally avoiding $T^+_{h(x)}$.

The key observation for handling case (C4) is:

▷ Claim 21. If (C4) holds, then:

(1) For any vertex $c \in T_{h(x)} \setminus \{y\}$, u and c are xy-disconnected.

(2) For any vertex $c \notin T_{h(x)} \cup \{x, y\}$, u and c are xy-connected iff they are $T^+_{h(x)}$-connected.

Proof. For (1), assume towards a contradiction that there exists a u-c path P in $G \setminus \{x, y\}$. Let c' be the first (closest to u) vertex from $T_{h(x)}$ appearing in P. Since $c' \neq y$, and the subpath $P[u, c']$ internally avoids $T^+_{h(x)}$, we get a contradiction to (C4).

The "if" direction of (2) is trivial as $\{x, y\} \subseteq T^+_{h(x)}$. For the "only if" direction, let P be a u-c path in $G \setminus \{x, y\}$. It suffices to prove that P avoids $T_{h(x)}$, but this follows directly from (1). ◁

We handle case (C4) as follows. If $v \in T_{h(x)}$, then by Claim 21(1) we determine that u, v are xy-disconnected, and we are done. Else, the child x_v of x on $T[x, v]$ is light, hence $x_v \in I(v)$. Therefore $\mathsf{L_{dep}}(v)$ contains $\mathsf{CID}(v, G \setminus T^+_{h(x)})$ (set $a = v$, $b = x$ and $b' = x_v$). As x_u is also light by (C4), we can find $\mathsf{CID}(u, G \setminus T^+_{h(x)})$ in a similar fashion. By Claim 21(2), comparing these CIDs allows us to determine the xy-connectivity of u, v, and we are done again.

If the corner case (C4) does not hold, then $\widehat{u}$ is indeed a valid xy-analog of u. Namely:

▷ Claim 22. If $\widehat{u}$ is defined, then (i) $\widehat{u} \in T_{h(x)}$, (ii) $u, \widehat{u}$ are xy-connected, and (iii) one can find the label $\mathsf{L_{AH}}(\widehat{u})$.

Proof. If x_u is heavy: Then (i) and (ii) are trivial. For (iii) we simply note that $\mathsf{L_{dep}}(u)$ stores $\mathsf{L_{AH}}(u)$.

If x_u is light and $\mathsf{AnSet}(u, x) \setminus \{y\} \neq \emptyset$: Then $\widehat{u} \in \mathsf{AnSet}(u, x) \setminus \{y\}$. By definition of $\mathsf{AnSet}(u, x)$ it holds that $\widehat{u} \in T_{h(x)}$, which gives (i), and that there is a u-$\widehat{u}$ path avoiding $T^+_{h(x)} \setminus \{\widehat{u}\}$, and consequently also $\{x, y\}$, which gives (ii). For (iii), we note that as x_u is light it holds that $x_u \in I(u)$. Thus, $\mathsf{L_{dep}}(u)$ stores the $\mathsf{L_{AH}}$ labels of the vertices in $\mathsf{AnSet}(u, x)$, and in particular stores $\mathsf{L_{AH}}(\widehat{u})$. ◁

Finalizing. We have shown a procedure that either determines directly the xy-connectivity of u, v, or certifies that $y \in T_{h(x)}$ and finds xy-analogs $\widehat{u}, \widehat{v} \in T_{h(x)}$ of u, v (respectively) along with their (AH)-labels $\mathsf{L_{AH}}(\widehat{u}), \mathsf{L_{AH}}(\widehat{v})$. In the latter case, we answer the (AH)-query $\langle \widehat{u}, \widehat{v}, x, y \rangle$ using the $\mathsf{L_{AH}}$ labels[10] and determine the xy-connectivity of $\widehat{u}, \widehat{v}$, or equivalently of u, v. This concludes the decoding algorithm for dependent failures. We therefore have:

▶ **Lemma 23.** *There exists $O(\log^3 n)$-bit labels $\mathsf{L_{dep}}$ supporting the dependent failures case.*

By combining the $\mathsf{L_{dep}}$ labels of Lemma 23 with the $\mathsf{L_{ind}}$ labels of Lemma 13, we obtain the 2-VFT labels of Theorem 2. The preprocessing time analysis is found in the full version.

References

1 Ittai Abraham, Shiri Chechik, and Cyril Gavoille. Fully dynamic approximate distance oracles for planar graphs via forbidden-set distance labels. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 1199–1218, 2012.

2 Ittai Abraham, Shiri Chechik, Cyril Gavoille, and David Peleg. Forbidden-set distance labels for graphs of bounded doubling dimension. *ACM Trans. Algorithms*, 12(2):22:1–22:17, 2016.

3 Surender Baswana, Keerti Choudhary, Moazzam Hussain, and Liam Roditty. Approximate single source fault tolerant shortest path. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1901–1915. SIAM, 2018.

4 Giuseppe Di Battista and Roberto Tamassia. Incremental planarity testing (extended abstract). In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 436–441. IEEE Computer Society, 1989.

[10] $\mathsf{L_{AH}}(x)$ and $\mathsf{L_{AH}}(y)$ are stored in $\mathsf{L_{dep}}(x)$ and $\mathsf{L_{dep}}(y)$ respectively.

5 Giuseppe Di Battista and Roberto Tamassia. On-line maintenance of triconnected components with spqr-trees. *Algorithmica*, 15(4):302–318, 1996.

6 Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. *CoRR*, abs/2203.00671, 2022. doi:10.48550/arXiv.2203.00671.

7 Joseph Cheriyan, Ming-Yang Kao, and Ramakrishna Thurimella. Scan-first search and sparse certificates: an improved parallel algorithm for k-vertex connectivity. *SIAM Journal on Computing*, 22(1):157–174, 1993.

8 Keerti Choudhary. An optimal dual fault tolerant reachability oracle. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPIcs*, pages 130:1–130:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

9 Bruno Courcelle and Andrew Twigg. Compact forbidden-set routing. In *STACS 2007, 24th Annual Symposium on Theoretical Aspects of Computer Science, Aachen, Germany, February 22-24, 2007, Proceedings*, pages 37–48, 2007.

10 Michal Dory, Yuval Efron, Sagnik Mukhopadhyay, and Danupon Nanongkai. Distributed weighted min-cut in nearly-optimal time. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 1144–1153. ACM, 2021.

11 Michal Dory and Merav Parter. Fault-tolerant labeling and compact routing schemes. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 445–455. ACM, 2021.

12 Ran Duan and Seth Pettie. Dual-failure distance and connectivity oracles. In *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*, pages 506–515. SIAM, 2009.

13 Ran Duan and Seth Pettie. Connectivity oracles for graphs subject to vertex failures. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 490–509, 2017.

14 Ran Duan and Seth Pettie. Connectivity oracles for graphs subject to vertex failures. *SIAM J. Comput.*, 49(6):1363–1396, 2020.

15 Maciej Duleba, Pawel Gawrychowski, and Wojciech Janczewski. Efficient labeling for reachability in directed acyclic graphs. In Yixin Cao, Siu-Wing Cheng, and Minming Li, editors, *31st International Symposium on Algorithms and Computation, ISAAC 2020, December 14-18, 2020, Hong Kong, China (Virtual Conference)*, volume 181 of *LIPIcs*, pages 27:1–27:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

16 Pawel Gawrychowski, Shay Mozes, and Oren Weimann. Minimum cut in $o(m \log^2 n)$ time. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 57:1–57:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

17 Loukas Georgiadis, Giuseppe F. Italiano, Luigi Laura, and Nikos Parotsidis. 2-vertex connectivity in directed graphs. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, volume 9134 of *Lecture Notes in Computer Science*, pages 605–616. Springer, 2015.

18 Loukas Georgiadis and Robert Endre Tarjan. Dominators, directed bipolar orders, and independent spanning trees. In Artur Czumaj, Kurt Mehlhorn, Andrew M. Pitts, and Roger Wattenhofer, editors, *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part I*, volume 7391 of *Lecture Notes in Computer Science*, pages 375–386. Springer, 2012.

19 Mohsen Ghaffari, Krzysztof Nowicki, and Mikkel Thorup. Faster algorithms for edge connectivity via random 2-out contractions. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 1260–1279. SIAM, 2020.

20 Mohsen Ghaffari and Goran Zuzic. Universally-optimal distributed exact min-cut. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*, pages 281–291. ACM, 2022.

21 Manoj Gupta and Shahbaz Khan. Multiple source dual fault tolerant BFS trees. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 127:1–127:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

22 Zhiyang He, Jason Li, and Magnus Wahlström. Near-linear-time, optimal vertex cut sparsifiers in directed acyclic graphs. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 52:1–52:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

23 John E. Hopcroft and Robert Endre Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2(3):135–158, 1973.

24 Arkady Kanevsky, Roberto Tamassia, Giuseppe Di Battista, and Jianer Chen. On-line maintenance of the four-connected components of a graph (extended abstract). In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 793–801. IEEE Computer Society, 1991. `doi:10.1109/SFCS.1991.185451`.

25 Sampath Kannan, Moni Naor, and Steven Rudich. Implicit representation of graphs. *SIAM Journal on Discrete Mathematics*, 5(4):596–603, 1992.

26 David R. Karger. Global min-cuts in rnc, and other ramifications of a simple min-cut algorithm. In Vijaya Ramachandran, editor, *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, pages 21–30. ACM/SIAM, 1993.

27 Neelesh Khanna and Surender Baswana. Approximate shortest paths avoiding a failed vertex: Optimal size data structures for unweighted graph. In *27th International Symposium on Theoretical Aspects of Computer Science-STACS 2010*, pages 513–524, 2010.

28 Jason Li, Danupon Nanongkai, Debmalya Panigrahi, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Vertex connectivity in poly-logarithmic max-flows. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 317–329. ACM, 2021.

29 Danupon Nanongkai, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Breaking quadratic time for small vertex connectivity and an approximation scheme. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 241–252. ACM, 2019.

30 Merav Parter. Dual failure resilient bfs structure. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 481–490, 2015.

31 Merav Parter. Small cuts and connectivity certificates: A fault tolerant approach. In *33rd International Symposium on Distributed Computing*, 2019.

32 Merav Parter. Distributed constructions of dual-failure fault-tolerant distance preservers. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 21:1–21:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

33 Merav Parter and Asaf Petruschka. Near-optimal distributed computation of small vertex cuts. In *36th International Symposium on Distributed Computing DISC*, 2022.

34 Seth Pettie and Longhui Yin. The structure of minimum vertex cuts. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 105:1–105:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

35 David Pritchard and Ramakrishna Thurimella. Fast computation of small cuts via cycle space sampling. *ACM Transactions on Algorithms (TALG)*, 7(4):46, 2011.

36 Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

37 Douglas B. West. *Introduction to Graph Theory*. Prentice Hall, 2 edition, September 2000.

A Sublinear f-VFT Labels

In this section, we provide an f-VFT labeling scheme with sublinear size for any $f = o(\log\log n)$. Note that labels of near-linear size are directly obtained by the f-EFT labeling scheme of [11] (e.g., by including in the labels of a vertex, the EFT-labels of all its incident edges). We show:

▶ **Theorem 24** (f-VFT Labels with Sublinear Size). *For every n-vertex graph $G = (V, E)$ and fixed parameter $f = o(\log\log n)$, there is a polynomial time randomized algorithm for computing f-VFT labels of size $\widetilde{O}(n^{1-1/2^{f-2}})$. For every query $\langle u, v, F\rangle$ for $F \subseteq V$, $|F| \leq f$, the correctness holds w.h.p.*

We use the EFT-labeling scheme of Dory and Parter [11], whose label size is independent in the number of faults. The correctness guarantee holds w.h.p. for a *polynomial* number of queries.

▶ **Theorem 25** (Slight Restatement of Theorem 3.7 in [11]). *For every undirected n-vertex graph $G = (V, E)$, there is a randomized EFT connectivity labeling scheme with labels $\mathsf{EL} : V \cup E \to \{0, 1\}^{\ell}$ of length $\ell = O(\log^3 n)$ bits (independent in the number of faults). For a given triplet along w the EL labels of $u, v \in V$ and every edge set $F \subseteq E$, the decoding algorithm determines, w.h.p., if u and v are connected in $G \setminus F$.*

The Labels. Our starting observation is that one can assume, w.l.o.g., that $|E(G)| \leq fn$ edges. This holds as it is always sufficient to apply the labeling scheme on the sparse f (vertex) connectivity certificate of G, which has at most fn edges, see e.g., [7]. The labeling scheme is inductive where the construction of f-VFT labels is based on the construction of $(f-1)$ VFT labels given by the induction assumption. For the base of the induction ($f = 2$), we use the 2-VFT labels of Theorem 2. The approach is then based on dividing the vertices into high-degree and low-degree vertices based on a degree threshold $\Delta = 2f \cdot n^{1-1/2^{f-2}}$. Formally, let V_H be all vertices with degree at least Δ. By our assumption, the number of high-degree vertices is at most $|V_H| \leq O(fn/\Delta)$. Letting $\mathsf{EL}(\cdot)$ denote that f-EFT labeling scheme of Theorem 25 by [11], the f-VFT label of v is given by Algorithm 5.

The Decoding Algorithm. Consider a query $\langle u, v, F\rangle \in V \times V \times V^{\leq f}$. We distinguish between two cases, based on the degrees of the faults F in the graph G. Assume first that there exists at least one high-degree vertex $x \in F \cap V_H$. In this case, the labels of every $w \in \{u, v\} \cup (F \setminus \{x\})$ includes the $(f-1)$ VFT label in $G \setminus \{x\}$, namely, $\mathsf{VL}_{\mathsf{f-1}}(v, G \setminus \{w\})$. We can then determine the F-connectivity of u,v using the decoding algorithm of the $(f-1)$-VFT labels (given by the induction assumption). It remains to consider the case where all

Algorithm 5 Construction of label $\mathsf{VL_f}(v)$ of for vertex v.

1 **store** $\mathsf{EL}(v)$;
2 **for** *each* $x \in V_H$ **do**
3   **store** $\mathsf{VL_{f-1}}(v, G \setminus \{x\})$;
4 **if** $v \in V \setminus V_H$ **then**
5   **store** $\mathsf{EL}(e = (u, v))$ for every adjecent edge $(u, v) \in G$;

vertices have low-degrees. In this case, the VFT-labels include the EFT-labels of u, v, and all failed edges, incident to the failed vertices. This holds as the label of every failed vertex $x \in F$ contains $\mathsf{EL}(e = (x, z))$ for each of its incident edges (x, z) in G. This allows us to apply the decoding algorithm of Theorem 24 in a black-box manner.

Label Size. We now turn to bound the label size. For every $f \leq n$, let $\sigma_V(f, n), \sigma_E(n)$ be an upper bound on f-VFT (resp., EFT) labels for n-vertex graphs. Assume by induction on $g \leq f - 1$ that

$$\sigma_V(g, n) = 2^{g-2} \cdot g \cdot n^{1-1/2^{g-2}} \cdot c \cdot \log^3 n \ , \tag{1}$$

where $c \cdot \log^3 n$ is the bound on the EFT labels of Theorem 25. This clearly holds for $g = 2$ (by Theorem 2). Denote the length of the f-VFT label for vertex v by $|\mathsf{VL_f}(v)|$. By taking $\Delta(f, n) = 2f \cdot n^{1-1/2^{f-2}}$ to be the degree threshold Δ in our f-VFT label construction, and using Eq. (1), we have:

$$|\mathsf{VL_f}(v)| \leq \sigma_V(f-1, n-1) \cdot (2nf/\Delta(f, n)) + \Delta(f, n) \cdot \sigma_E(f, n) \ \leq \sigma_V(f, n) \ .$$

This satisfies the induction step and provides a bound of $\sigma_V(f, n) = \widetilde{O}(n^{1-1/2^{f-2}})$ for every $f = o(\log \log n)$, as desired. Theorem 24 follows.

Safe Permissionless Consensus

Youer Pu
Cornell University, Ithaca, NY, USA

Lorenzo Alvisi
Cornell University, Ithaca, NY, USA

Ittay Eyal
Technion, Haifa, Israel

Abstract

Nakamoto's consensus protocol works in a permissionless model, where nodes can join and leave without notice. However, it guarantees agreement only probabilistically. Is this weaker guarantee a necessary concession to the severe demands of supporting a permissionless model? This paper shows that, at least in a benign failure model, it is not. It presents Sandglass, the first permissionless consensus algorithm that guarantees deterministic agreement and termination with probability 1 under general omission failures. Like Nakamoto, Sandglass adopts a *hybrid synchronous* communication model, where, at all times, a majority of nodes (though their number is unknown) are correct and synchronously connected, and allows nodes to join and leave at any time.

2012 ACM Subject Classification Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Consensus, Permissionless, Nakamoto, Deterministic Safety

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.33

Related Version *Full Version*: `https://eprint.iacr.org/2022/796.pdf` [25]

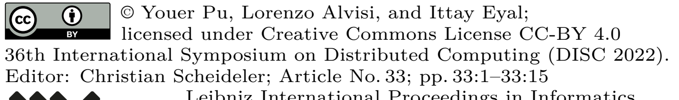
Funding This work was supported in part by the NSF grant CNS-CORE 2106954, BSF and IC3.

1 Introduction

The publication of Bitcoin's white paper [22], besides jumpstarting an industry whose market is expected to reach over $67B by 2026 [27], presented the distributed computing community with a fundamental question [12]: how should the agreement protocol at the core of Nakamoto's blockchain construction (henceforth, *Nakamoto's Consensus* or *NC*) be understood in light of the combination of consensus and state machine replication that the community has studied for over 30 years? The similarities are striking: in both cases, the goal is to create an append-only distributed ledger that everyone agrees upon, which NC calls a *blockchain*. But so are the differences. Unlike traditional consensus algorithms, where the set of participants n is known and can only be changed by running an explicit reconfiguration protocol, Nakamoto's consensus is *permissionless*: it does not enforce access control and allows the number and identity of participants to change without notice. It only assumes that the computing power of the entire system is bounded, which effectively translates to assuming the existence of an upper bound $\mathcal{N}$ on the number of participants. [1]

To operate under these much weaker assumptions, NC adopts a new mechanism for reaching agreement: since the precise value of n is unknown, NC forsakes explicit majority voting and relies instead on a *Proof of Work* (PoW) lottery mechanism [22], designed to

[1] The bound can be trivial, e.g., equal to the number of atoms in the Universe, but it needs to exist; otherwise, if it would be possible for a large, unknown group of nodes to be secretly adding blocks onto a different branch of the blockchain, and Nakamoto's decisions would never be, even probabilistically, safe.



© Youer Pu, Lorenzo Alvisi, and Ittay Eyal;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 33; pp. 33:1–33:15
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

drive agreement towards the blockchain whose construction required the majority of the computational power of all participants. Finally, whereas traditional consensus protocols guarantee agreement deterministically, NC can do so only probabilistically; furthermore, that probability approaches 1 only as termination time approaches infinity. Is settling for these weaker guarantees the inevitable price of running consensus in a permissionless setting?

In this paper we show that, at least in a benign failure model, one can do much better. We present *Sandglass*, a permissionless consensus algorithm that guarantees *deterministic* agreement and terminates with probability 1. It operates in a model based on Nakamoto's. Our model allows an arbitrary number of participants to join and leave the system at any time and stipulates that at no time the number of participants exceeds an upper bound $\mathcal{N}$ (though the *actual* number n of participants at any given time is unknown). Further, like Nakamoto's, it is *hybrid synchronous*, in that, at all times, a majority of participants are correct and able to communicate synchronously with one another. We call these participants *good*; our protocol's safety and liveness guarantees apply to them. Participants that are not good (whether because they crash, perform omission failures, and/or experience asynchronous network connections) we call *defective*. Sandglass proceeds in asynchronous rounds, with a structure surprisingly reminiscent of Ben-Or's classic consensus protocol [3]. Let's review it. Nodes propose a value by broadcasting it; in the first round, each node proposes its initial value; in subsequent rounds, nodes propose a value chosen among those received in the previous round. Values come with an associated priority, initialized to 0. The priority of v depends on the number of consecutive rounds during which v was the only value received by the node proposing v – whenever a node receives a value other than v, it resets v's priority back to 0. When proposing a value in a given round, node p selects the highest priority value received in the previous round; if multiple values have the same priority, then it selects randomly among them. A node can safely decide a value v after sufficiently many consecutive rounds in which the proposals it receives unanimously endorse v (i.e., when v's priority is sufficiently high); and termination follows from the non-zero probability that the necessary sequence of unanimous, consecutive rounds will actually eventually occur.

Of course, embedding this structure in a permissionless setting introduces unprecedented challenges. Consider, for example, how nodes decide. In Ben-Or, a node decides v after observing two consecutive, unanimous endorsements of v; it can do so safely because any two majority sets of its fixed set of n nodes intersect in at least one correct node. This approach is clearly no longer feasible in a permissionless setting, where n is unknown and the set of nodes can change at any time.

Instead, Sandglass's approach to establish safety is inspired by one of the key properties of Nakamoto's PoW: whatever the value of n, whatever the identity of the nodes participating in the protocol at any time, the synchronously connected majority of *good nodes* will, in expectation, be faster than the remaining nodes in adding a new block to the blockchain.

Think now of adding a block b at position i of the blockchain as implicitly starting a new round of consensus for all the chain's positions that precede i; for each position, the new round proposes the corresponding block in the hash chain that ends at b. In this light, the greater speed in adding blocks that PoW promises to the majority of connected nodes translates into these nodes moving faster from one asynchronous round to the next in each of the consensus instances.

This insight suggests an alternative avenue for achieving deterministic consensus among good nodes – without relying on quorum intersection. Node p should decide on a value v only after it has seen v unanimously endorsed for sufficiently many rounds that, if p is good, the lead p (and all other good nodes) have gained over any defective node q proposing some other value is so large that q's proposals can no longer affect the proposal of good nodes.

Why can't the same approach be used to achieve deterministic consensus in Nakamoto's original protocol? Because Nakamoto's PoW mechanism, notwithstanding its name, is an indirect and imperfect vehicle for proving work. As evidence of performed work, Nakamoto presents the solution to a puzzle: this solution, however, could just have been produced as a result of a lucky guess. Thus, however unlikely, it is always possible in NC for defective nodes proposing a value other than v to catch up with, or even overtake, good nodes and reverse their decisions.

To avoid this danger, Sandglass relies on a different PoW mechanism, which ties the ability to propose a value to a *deterministic* amount of work. In particular, Sandglass nodes can propose a value in any round other than the first only after they have received a specific threshold of messages from the previous round. Therefore, each proposed value implicitly represents all the work required to generate the messages needed to clear the threshold. The threshold value is chosen as a function of the upper bound $\mathcal{N}$ on the number of nodes that at any time run the protocol, in such a way that, whatever is their actual number n, any node that does not receive messages from good nodes will inevitably take longer than them in moving from round to round.

The full power of this PoW mechanism, however, comes from pairing it with the idea, which we borrow from Ben Or, of associating a priority with the values being proposed. With a fixed set of n nodes, Ben Or leverages priorities and quorum intersection to safely decide a value v once it has reached priority 2, because it can guarantee that henceforth every node executing in the same round as a correct node will propose v. In a permissionless setting, we show that the combination of priorities and our PoW mechanism allows Sandglass to offer good nodes the same guarantee (though, as we will see, v will be required to reach a significantly higher priority value!). Intuitively, by the time v reaches the priority necessary to decide, any node q that manages not to fall behind (and thus become irrelevant) to the unanimous majority of good nodes who have kept proposing v must have received *some* of the messages proposing v from some good nodes. Furthermore, to keep up, q must have received such messages often enough that, given how the priority of received values determines what a node can propose, it would be impossible for q to propose any value other than v.

In summary, this paper makes the following contributions: (*i*) it formalizes Nakamoto's permissionless model in the vocabulary of traditional consensus analysis; (*ii*) it introduces novel proof strategies suitable for this new model; (*iii*) it exposes the connection between PoW and a voting mechanism that can be implemented by message passing; and (*iv*) it introduces Sandglass, the first protocol that achieves deterministic agreement in a permissionless setting under hybrid synchrony.

2 Related work

The consensus problem has been studied for decades, covering both benign and Byzantine faults under different synchrony assumptions. Common across these classic works is the assumption that the set of nodes that participate in running the protocol is either constant or changes through an agreement among the incumbents (*reconfiguration*). In contrast, Sandglass allows for participants to change arbitrarily and without any coordination, as long as at all times a majority of nodes is correct and synchronously connected. More recent papers also explore models where participants can change dynamically at any time, subject to guarantees of well-behaved majority; unlike Sandglass, those works achieve only probabilistic safety guarantees. We briefly review related prior work in more detail below.

The permissionless nature of our model implies that consensus solutions for classical models (e.g., [13]) do not apply. For synchronous networks, previous solutions rely on the fact that the number of failures is bounded in a period of time. They tolerate up to $(n-1)$ benign failures [29] or Byzantine failures with authentication [7, 18]. For an asynchronous network, Fisher, Lynch, and Paterson [8] show that it is impossible to solve consensus with deterministic safety and liveness even with a single crash failure. Various protocols (e.g., [16, 26, 28]) thus either solve asynchronous consensus with weaker liveness guarantees than deterministic termination, or provide deterministic termination after a Global Stabilization Time (GST) (e.g., [4]). They use logical rounds, and for each round collect messages from a sufficient number of (authenticated) nodes, tolerating fewer than $\frac{n}{2}$ failures in a benign failure model [3, 17], and $\frac{n}{3}$ failures with Byzantine failures and authentication [4, 31]. Although our model is not directly comparable, we note that our protocol matches the $(n/2)$ bound of a benign model in an asynchronous network, despite assuming synchrony among good nodes.

Aspnes et al. [2] explore the consensus problem in an asynchronous benign model where an unbounded number of nodes can join and leave [9], but where at least one node is required to live forever, or until termination. It is easy to see that in their model, but without this latter assumption, deterministic safety is impossible. In contrast, Sandglass, in a hybrid synchronous model, guarantees deterministic safety while allowing *all* nodes to freely join and leave.

Consider two groups of good nodes with different initial values running from $t=0$, with messages within the groups delivered immediately, but messages between the two groups are delayed until at least one in each group decides. By validity of consensus, the two groups will decide on different values, which violates agreement.

A newer line of work, starting with Nakamoto [22], studies systems where principals can unilaterally join or leave without notifying previous participants. These protocols (e.g., [21, 30]) are based on probabilistic assumptions and provide probabilistic guarantees. Specifically, participation is based on probabilistic proofs of work, and the assumption that no minority can find most proofs of work in a long period. They provide safety with high probability, given a sufficiently long running time (latency) [6, 15, 23, 10]. Nonetheless, they are all based on probabilistic techniques and provide probabilistic guarantees, which cannot be directly translated to deterministic guarantees.

Several protocols, inspired by the PoW approach, achieve consensus among a large group of principals while requiring the active participation of only a subset of them. In the Sleepy Model [24] participants join and leave ("sleep"); the assumptions and guarantees of the consensus protocol presented for this model are as probabilistic as those of pure proof of work. Momose and Ren [20] present a consensus protocol in the Sleepy Model with constant latency and deterministic agreement; however, their protocol does not guarantee progress until the participation is stable. Ouroboros [14, 5] forms a chain in the spirit of PoW but using internal tokens for the random choice of participants, again leading to probabilistic guarantees. In Algorand [11], committees elect one another in a series of reconfigurations, with assumptions and guarantees similar to classical consensus, except that participants are chosen at random from a large pool, with a negligible probability of a Byzantine majority – again, providing probabilistic guarantees.

In contrast, despite its permissionless model, our protocol guarantees deterministic safety and terminates with probability 1. We note other differences: Sandglass's failure model assumes only benign failure and is thus stronger than the Byzantine model adopted by many of these works, but its network assumptions are weaker, as defective nodes can experience asynchronous communication, and all nodes can join or leave instantaneously.

Abraham and Malkhi [1] formalize Nakamoto's Consensus within a classical disturbed systems framework, and in particular abstract the PoW primitive as a Pre-Commit, Non-Equivocation, Leader Election (PCNELE) Oracle. However, the power of this probabilistic oracle is similar to that of Nakamoto's PoW and yields a consensus protocol that provides only probabilistic guarantees as Nakamoto.

Lewis-Pye and Roughgarden [19] show that deterministic consensus cannot be achieved in a permissionless synchronous model with Byzantine nodes, let alone in a partially synchronous model (where communication becomes synchronous only after some point unknown to the processes). Sandglass shows, for the first time, that deterministic safety and termination with probability 1 can be achieved in a permissionless model, though the network is hybrid-synchronous rather than synchronous. The exploration of a Byzantine model remains for future work.

3 Model

The system comprises an infinite set of nodes $p_1, p_2, \ldots$. Time progresses in discrete steps; in each step, a subset of the nodes is *active* and the rest are *inactive*. At each step, active nodes are partitioned into *good* and *defective* subsets.

We assue a hybrid synchronous model. *Good* nodes are correct, and the network that connects them to one another is synchronous; at all times, a majority of active nodes are good. *Defective* nodes may suffer from benign failures, such as crashes and omission failures, or simply lack a synchronous connection with some good node.

The system progress is orchestrated by a *scheduler*. In each step, the scheduler can activate any inactive node p_i (we say that p_i has *joined* the system) and deactivate an active node (which then *leaves* the system). The scheduler chooses which nodes to activate and deactivate arbitrarily, subject only to the following three constraints: (i) The upper bound of active nodes in any step is $\mathcal{N}$; (ii) there is at least one active node in every step; and (iii) in every step the majority of active nodes is good.

In each step where it is active, each node p_i executes the stateful protocol shown as procedure *Step* in Sandglass's pseudocode (see Algorithm 1). It can execute computations, update its state variables, and communicate with other nodes with a broadcast network. In particular, since Sandglass assumes benign failures, every active node, whether good or defective, waits for a full step to elapse before sending its next message.

The network allows each active node to *broadcast* and *receive* unauthenticated messages. Node p_i broadcasts a message m with a $\mathit{Broadcast}_i(m)$ instruction and receives messages broadcast by itself and others with a $\mathit{Receive}_i$ instruction. The network does not generate or duplicate messages, *i.e.*, if in step t a node p_i receives message m with $\mathit{Receive}_i$, then m was sent in some step $t' < t$.

The communication model is designed to capture the design of Nakamoto's consensus, which relies on an underlying network layer to propagate and store blocks. Nakamoto's network layer provides a shared storage of data structures, called blocks, and guarantees delivery of published blocks within a bounded time. Each block includes cryptographically secure references to all blocks seen by its creator. This allows a newly joined node to receive and validate the entire history of published blocks. Thus, in our model, the scheduler determines when each message is delivered to each node under the following constraints.

First, propagation time is bounded between any pair of good nodes. Formally: if a good node p_i calls $\mathit{Broadcast}_i(m)$ in step t, and if a good node p_j calls $\mathit{Receive}_j$ in step $t' > t$, then m is returned, unless it was already received by p_j in an earlier call to $\mathit{Receive}_j$. Thus, a newly activated good node is guaranteed, upon executing its first *Receive*, to receive all messages from other good nodes broadcast in the steps prior to its activation.

Second, the network is reliable, but there is no delivery bound unless both nodes are good. Formally: For any two nodes p_i and p_j, where at least one of p_i and p_j is defective, and for a message m broadcast by p_i, if node p_j calls $Receive_j$ infinitely many times, then m is eventually delivered.

Each node is initiated when joining the system with an initial value $v_i \in \{\mathsf{a}, \mathsf{b}\}$. An active node p_i can decide by calling a $Decide_i(v)$ instruction for some value v. The goal of the nodes is to reach a consensus based on these values:

▶ **Definition 1** (Agreement). *If a good node decides a value v, then no good node decides a value other than v.*

▶ **Definition 2** (Validity). *If all nodes that ever join the system have initial value v and any node (whether good or defective) decides, then it decides v.*

▶ **Definition 3** (Termination). *Every good node that remains active eventually decides.*

4 Protocol

To form an intuition for the mechanics of Sandglass, it is useful to compare and contrast it with Ben-Or. From a distance, the high-level structure of the two protocols is strikingly similar: execution proceeds in asynchronous rounds; progress to the next round depends on collecting a threshold of messages sent in the current round; safety and liveness depend on the correctness of a majority of nodes; and nodes decide a value v when, for sufficiently many consecutive rounds, all the messages they collect propose v. But looking a little closer, the differences are equally striking. On the one hand, Sandglass's notion of node correctness and its hybrid synchronous model are stronger than Ben-Or's. Sandglass assumes a majority of good nodes that are not only free from crashes and omissions, but also synchronously connected to one another. On the other hand, in Sandglass, unlike Ben-Or, the number n of nodes running the protocol is not only unknown, but may be changing all the time. These differences motivate four key aspects that separate the two protocols:

Choosing a threshold In Ben-Or, a node advances to a new round only after having received a message from a majority of nodes. This strict condition for achieving progress is critical to how Ben-Or establishes Agreement. Any node that, from a majority of the nodes in round r, receives a set of messages that unanimously propose v, can be certain that (i) there cannot exist in r also a unanimous majority proposing a value other than v and (ii) no node can proceed to round $(r+1)$ unaware that v is among the values proposed in round r. Nodes that isolate themselves from a majority simply do not make any progress; and since all majority sets intersect, nodes cannot make contradictory decisions.

Unfortunately, this approach is unworkable in Sandglass: when the cardinality and membership of the majority set can change at any time, receiving messages from a majority can no longer serve as a binary switch to trigger progress. More generally, thresholds based on the cardinality of the set of nodes from which one receives messages become meaningless. Instead, Sandglass allows nodes to broadcast multiple messages during a round, one in each of the round's steps, and lets nodes move to round $(r+1)$ once they have collected a specified threshold of messages sent in round r.

Think of the threshold $\mathcal{T}$ of messages that allows a node to move to a new round as the number of grains of sands in a sandglass: a node (figuratively) flips the sandglass at the beginning of a round, and cannot move to the next until all $\mathcal{T}$ sand grains have moved to the bottom bulb. The value of $\mathcal{T}$ is the same for all nodes; the speed at which messages are collected, however – the width of their sandglass's neck – is not, and can change from

step to step: if all nodes broadcast messages at the same rate, the larger the number of nodes that one receives messages from in a timely fashion, the faster it will be to reach the threshold. Thus, while in Sandglass setting a threshold cannot altogether prevent nodes that don't receive messages from a majority from making progress, it ensures that they will progressively fall behind those who do.

Exchanging messages In each step of the protocol, a node currently in round r (i) determines, on the basis of the messages received so far, what is the largest round $r_{max} \geq r$ for which it has received the required threshold of messages and (ii) broadcasts a message for round r_{max}, which includes the node's current proposed value, as well as the critical *metadata* discussed below.

Keeping history Unlike Ben-Or, Sandglass allows nodes to join the system at any time. To bring a newly activated node up to speed, each message broadcast by a node p in round r carries a *message coffer* that includes (i) the set of messages (at least $\mathcal{T}$ of them) p collected in round $r-1$ to advance to round r; (ii) recursively, the set of messages in those messages' coffers; and (iii) the set of messages p collected so far for round r.

Respecting priority In Ben-Or, a node decides v if, for two consecutive rounds, v is the only value it collects from a majority set. To ensure the safety of that decision, Ben-Or assigns a *priority* to the value v that a node p proposes: if v was unanimously proposed by all the messages p collected in the previous round, it is given priority 1; otherwise, 0. Nodes that collect more than one value in round r, propose for round $r+1$ the one among them with the highest priority, choosing by a coin flip in the event of a tie. Sandglass uses a similar idea, although its different threshold condition requires a much longer streak of consecutive rounds where v is unanimously proposed before v's priority can be increased. To keep track of the length of that streak, every message sent in a given round r carries a *unanimity counter*, which the sender computes upon entering r.

4.1 Selecting the Threshold

Unlike Ben-Or, Sandglass's threshold condition can not altogether prevent nodes from making progress. It is perhaps surprising that, by leveraging only the assumption that at all times a majority of nodes are good (i.e., correct and synchronously connected with each other) without ever knowing precisely how many they actually are, Sandglass retains enough of the disambiguating power of intersecting majorities to ultimately yield deterministic agreement.

In essence, Sandglass succeeds by causing defective nodes that isolate themselves from the majority of nodes in the systen to fall eventually so far behind that they no longer share the same round with good nodes. At the same time, it ensures that, once some good node has decided on a value v, nodes that manage to keep pace with good nodes will never propose anything other than v.

Of course, to obtain this outcome it is critical to set $\mathcal{T}$ appropriately. Consider two nodes, one good and one defective, and suppose they flip their sandglass at the same time – *i.e.*, they enter a new round in the same step. We want that, independent of how the number of active nodes may henceforth vary at each step, if the defective node only receives messages from other defective nodes (*i.e.*, if it fails to hear from a majority of nodes), it will reach the threshold $\mathcal{T}$ at least one step later than the good node will. The following lemma shows that setting $\mathcal{T}$ to $\lceil \frac{\mathcal{N}^2}{2} \rceil$ (where $\mathcal{N}$ is the upper bound on the maximum number of nodes active in any step) does the trick.

▶ **Lemma 4.** *For any k, consider any time interval comprising $(k+1)$ consecutive steps. Let the number of messages generated by good nodes and defective nodes in each step of the interval be, respectively, $g_0, g_1, ..., g_k$ and $d_0, d_1, ..., d_k$. Setting the threshold $\mathcal{T}$ to $\lceil \frac{\mathcal{N}^2}{2} \rceil$ ensures that, if $\Sigma_{i=1}^{i=k-1} g_i < \mathcal{T}$, then $\Sigma_{i=0}^{i=k} d_i < \mathcal{T}$.*

Proof. Note how the lemma does not count the messages generated by good nodes in the steps at the two ends of the interval. Recall that moving from the current round to the next requires a node to receive at least a threashold $\mathcal{T}$ of messages sent in the current round. Thus, we drop good messages from step 0 because good nodes that in step 0 enter a new round r are unable to count against the threshold for round r messages generated by good node that in step 0 are still in round $r-1$. And we similarly drop step k because good nodes may only need one of the messages sent by good nodes in step k to move to a new round – and have no use for the remaining messages in g_k.

We begin by observing that, when k is either 0 or 1, the lemma trivially holds, since in all steps defective nodes generate fewer than $\mathcal{N}$ messages. For example, when $k = 1$, $d_0 + d_1 < \frac{\mathcal{N}}{2} + \frac{\mathcal{N}}{2} = \mathcal{N} \leq \lceil \frac{\mathcal{N}^2}{2} \rceil$. We then prove the lemma for $k \geq 2$.

Let $\bar{g} = \frac{\Sigma_{i=1}^{i=k-1} g_i}{k-1}$ and $\bar{d} = \frac{\Sigma_{i=1}^{i=k-1} d_i}{k-1}$ denote, respectively, the average number of messages generated by good nodes and by defective nodes during the $k-1$ steps that include all but the interval's first and last step. Expressed in terms of $\bar{g}$ and $\bar{d}$, the lemma requires us to show that, if $\bar{g} \cdot (k-1) < \mathcal{T}$, then $\Sigma_{i=0}^{i=k} d_i = d_0 + \bar{d} \cdot (k-1) + d_k < \mathcal{T}$ when $\mathcal{T}$ is chosen to equal $\lceil \frac{\mathcal{N}^2}{2} \rceil$.

Assume $\bar{g} \cdot (k-1) < \mathcal{T}$; then $k - 1 < \frac{\mathcal{T}}{\bar{g}}$. Substituting for $(k-1)$ in the formula that computes the messages generated by defective nodes, we have:

$$
\begin{aligned}
\Sigma_{i=0}^{i=k} d_i &= \bar{d} \cdot (k-1) + d_0 + d_k \\
&< \bar{d} \cdot \frac{\mathcal{T}}{\bar{g}} + d_0 + d_k \quad (\text{since } (k-1) < \frac{\mathcal{T}}{\bar{g}}) \\
&\leq \bar{d} \cdot \frac{\mathcal{T}}{\bar{g}} + \frac{\mathcal{N}-1}{2} + \frac{\mathcal{N}-1}{2} \quad (\text{since defective nodes are always a minority}) \\
&\leq \bar{d} \cdot \frac{\mathcal{T}}{\bar{g}} + \frac{\mathcal{T}}{\frac{\mathcal{N}^2}{2}}(\mathcal{N}-1) \quad (\text{since } \mathcal{T} = \lceil \frac{\mathcal{N}^2}{2} \rceil \geq \frac{\mathcal{N}^2}{2}) \\
&= \mathcal{T}(\frac{\bar{d}}{\bar{g}} + \frac{2(\mathcal{N}-1)}{\mathcal{N}^2}).
\end{aligned}
$$

Then, to establish that $\Sigma_{i=0}^{i=k} d_i < \mathcal{T}$, it suffices to prove that $\frac{\bar{d}}{\bar{g}} + \frac{2(\mathcal{N}-1)}{\mathcal{N}^2} < 1$.

Since for any i, $d_i \leq g_i - 1$ and $d_i + g_i \leq \mathcal{N}$, we know that $\bar{d} \leq \bar{g} - 1$ and $\bar{d} + \bar{g} \leq \mathcal{N}$. Dividing both inequalities by $\bar{g}$ yields $\frac{\bar{d}}{\bar{g}} \leq \min(1 - \frac{1}{\bar{g}}, \frac{\mathcal{N}}{\bar{g}} - 1)$. Note that the largest value of $\min(1 - \frac{1}{\bar{g}}, \frac{\mathcal{N}}{\bar{g}} - 1)$ occurs when $1 - \frac{1}{\bar{g}} = \frac{\mathcal{N}}{\bar{g}} - 1$; solving for $\bar{g}$ and plugging the solution back in, gives us: $\min(1 - \frac{1}{\bar{g}}, \frac{\mathcal{N}}{\bar{g}} - 1) \leq \frac{\mathcal{N}-1}{\mathcal{N}+1}$.

Therefore, we have that $\frac{\bar{d}}{\bar{g}} + \frac{2(\mathcal{N}-1)}{\mathcal{N}^2} \leq \frac{\mathcal{N}-1}{\mathcal{N}+1} + \frac{2(\mathcal{N}-1)}{\mathcal{N}^2} = \frac{\mathcal{N}^3+\mathcal{N}^2-2}{\mathcal{N}^3+\mathcal{N}^2} < 1$. ◀

4.2 Protocol Mechanics

Protocol 1, besides showing how Sandglass initializes its key variables, presents the code that node p_i executes to take a step. Every step begins with adding all received messages, as well as the messages in their message coffers, to a single set, Rec_i (lines 4 - 5). Going over the elements of that set, p_i determines the largest round r_{max} for which it has received at least a threshold $\mathcal{T}$ of messages, and, if the condition at line 6 holds, sets the current round to $(r_{max} + 1)$ (line 7). Upon entering a new round, p_i does four things. First, after resetting

Algorithm 1 Sandglass: Code for node p_i.

```
1:  procedure Init(input_i)
2:      v_i ← input_i; priority_i ← 0; uCounter_i ← 0; r_i = 1; M_i = ∅; Rec_i = ∅; uid_i = 0
3:  procedure step
4:      for all m = (·,·,·,·,·,M) received by p_i do
5:          Rec_i ← Rec_i ∪ {m} ∪ M
6:      if max_{|Rec_i(r)|≥T}(r) ≥ r_i then
7:          r_i = max_{|Rec_i(r)|≥T}(r) + 1
8:          M_i = ∅
9:          for all m = (·, r_i − 1,·,·,·,M) ∈ Rec_i(r_i − 1) do
10:             M_i ← M_i ∪ {m} ∪ M
11:         Let C be the multi-set of messages in M_i(r_i − 1) with the largest priority.
12:         if all messages in C have the same value v then
13:             v_i ← v
14:         else
15:             v_i ← one of{a, b}, chosen uniformly at random
16:         if all messages in M_i(r_i − 1) have the same value v_i then
17:             uCounter_i ← 1 + min{uCounter|(·, r_i − 1, v_i,·, uCounter,·) ∈ M_i(r_i − 1)}
18:         else
19:             uCounter_i ← 0
20:         priority_i ← max(0, ⌊uCounter_i / T⌋ − 5)
21:         if priority_i ≥ 6T + 4 then
22:             Decide_i(v_i)
23:     uid_i ← uid_i + 1;
24:     M_i ← M_i ∪ Rec_i(r_i)
25:     broadcast (p_i, uid_i, r_i, v_i, priority_i, uCounter_i, M_i)
```

its message coffer M, p_i collects in the coffer all the messages it received from the previous round – as well as the messages stored in the coffers of those messages (lines 8 - 10). Second, p_i chooses the value v that it will propose in the current round (lines 11 - 15): it picks the highest-priority value among those collected in its coffer for the previous round ; if more than one value qualifies, it chooses among them uniformly at random. Third, p_i computes the unanimity counter and the priority for all messages that p_i will broadcast during the current round (lines 16 -20). The counter represents, starting from the previous round and going backwards, the longest sequence of rounds for which all corresponding messages in p_i's coffer unanimously proposed v. The priority is simply a direct function of the value of the unanimity counter: we maintain it explicitly because it makes it easier to describe how Sandglass works. Finally, if v's priority is high enough, p_i decides v (lines 21- 22). Whether or not it starts a new round, p_i ends every step by broadcasting a message (line 25): before it is sent, the message is made unique (line 23) and p_i adds to the message's coffer all messages received for the current round (line 24).

5 Correctness: Overview

Sandglass upholds the definitions of Validity, Agreement, and Termination (with probability 1) given in Section 3. We overview the proof below, as its approach differs from proofs of classical, permissioned protocols. We defer the proof to the full version of this report [25], which includes the formal statements of the lemmas we informally state below.

Validity is easily shown by induction on the round number, since if all nodes that join have the same value, there is only one value that can be sent in each round. Establishing Agreement and Termination is significantly more involved, and hinges on a precise understanding of the kinematics of good and defective nodes – and how that interacts with the ability of good nodes to converge on decision value and on the number of rounds necessary to do so safely. How clustered are good nodes as they move from round to round? At what rate do good nodes gain ground over defective nodes that cut themselves out from receiving messages from good nodes? How often do defective nodes need to receive messages from good nodes to be in turn able to have their messages still be relevant to good nodes?

The answer to these and similar questions constitute the scaffolding of lemmas and corollaries on which the proofs of Agreement and Termination rely. We discuss it in greater detail below, before moving on to the proofs.

5.1 The Scaffolding

The protocol achieves several properties that facilitate the consensus proof.

First, it keeps good nodes close together as they move from round to round. Specifically, the following lemmas hold:

▶ **Lemma 5.** *In any step two good nodes are at most one round apart.*

▶ **Lemma 6.** *If in any step a good node is in round r, then by the next step all good nodes are guaranteed to be at least in round r.*

A key intuition that guides the proof is that defective nodes move from round to round slower than good ones. There is a complication, however: this intuition holds only when defective nodes receive messages only from their own kind; in fact they can actually advance *faster* than good ones by combining messages from good nodes with messages from defective nodes that do not reach the good nodes. Nonetheless, we can show that

▶ **Lemma 7.** *At any step a defective node is at most one round ahead of any good node.*

Second, the protocol guarantees information sharing among good nodes. This may appear trivial to establish, since good nodes are correct and synchronously connected, but the *laissez-faire* attitude of the permissionless model, with nodes joining and leaving without coordination at any step, complicates matters significantly, making it impossible to prove seemingly basic properties. For example, consider a good node p that, in round r and step T, proposes a value v with a positive *uCounter*. It would feel natural to infer that all good nodes must have proposed v in the previous round – but it would also be wrong. If p just entered r in step T, it would in fact ignore any value proposed by good nodes that newly joined the systems in step T, but are still in round $r-1$. Fortunately, we show that a much weaker form of information sharing among good nodes is sufficient to carry the day. As a matter of terminology, let's say that a node *collects* a message in a round if it receives the message and does not ignore it (messages originated from a lower round number are ignored). We then prove the following facts about collected messages:

▶ **Lemma 8.** *In any round, a good node collects at least one message from a good node.*

▶ **Corollary 9.** *For any round, there exists a message from a good node that is collected by all good nodes.*

Third, it allows us to establish the basis for a key insight about the kinematics of Sandglass nodes that will be crucial for proving Agreement and Termination: in the long run, the only values proposed by defective nodes that remain relevant to the outcome of consensus are those that have been, in turn, recently influenced by values proposed by good nodes. This insight stands on a series of intermediate results. We already saw (Lemma 4) that, given any sequence of steps, if good nodes cannot generate enough messages to get into the next round, neither can defective nodes, even if they, unlike good nodes, are allowed to count messages generated in the two steps at the opposite ends of the period. It follows that

▶ **Lemma 10.** *During the steps that good nodes spent in a round, defective nodes can generate fewer than the* $\mathcal{T}$ *messages necessary to move to the next round.*

It all ultimately leads to the following lemma, which quantifies the slowdown experienced by defective nodes that don't allow themselves to be contaminated by good nodes:

▶ **Lemma 11.** *Defective nodes that do not collect any message from good nodes for* $k\mathcal{T}$ *consecutive rounds fall behind every good node by at least* $(k-1)$ *rounds.*

5.2 Agreement

The intuition behind our proof of Agreement is simple. To each value v proposed and collected by Sandglass nodes is associated a *uCounter*, which records the current streak of consecutive rounds for which all the messages collected by the proposer of v were themselves proposing v. Once v's *uCounter* reaches a certain threshold, v's *priority* increases; and once the value v proposed by a node reaches a given priority threshold, then a node decides v (see Algorithm 1, line 21). Since, as we saw, good nodes share information from round to round (recall Corollary 9), proving Agreement hinges on showing that, once a good node decides v, no good node will ever propose a value other than v. To prove that, we must in turn leverage what we learned about the kinematics of Sandglass nodes to identify a priority threshold that makes it safe for good nodes to decide. It should be large enough that, after it is reached, it becomes impossible for a defective node to change the proposal value of any good node.

The technical core of the Agreement proof then consists in establishing the truth of the following claim:

▷ Claim 12. Let p_d be the earliest good node to decide, in round r_d at step T_d. Suppose p_d decides v_d. Then, any good node p_g that in any step (whether before, at, or after T_d) finds itself in a round r_g that is at least as large as r_d, proposes v_d for r_g with *priority* at least 1. [2]

It is easy to see that if the above claim holds, then Agreement follows. Say that T_d is the earliest step in which a good node p_d, currently in round r_d, decides v_d. The claim immediately implies that no good node can decide a value other than v_d in a round greater or equal to r_d, since, from r_d on, every good node proposes v_d. Recall that, since good nodes are never more that one round apart at any step (Lemma 5), the earliest round a good node can find itself at T_d is (r_d-1); and that, by Lemma 6, every good node is guaranteed to be at least in round r_d by step (T_d+1). All that is left to show then is that no good node p', which at T_d found itself in round (r_d-1), can decide some value v' other than v_d. To this end, we leverage the information sharing that we proved exists among good nodes.

[2] Although proving Agreement does not require that v_d be proposed with priority at least 1, it makes proving the claim easier.

By Corollary 9, there is at least a message m generated in round $(r_d - 2)$ by a good node that is collected by all the good nodes. Since p_d at T_d has reached the priority threshold required to decide v_d, m must have proposed v_d; but if so, it would be impossible for good node p', which also must have collected m, to have reached the priority threshold required to decide a different value v'.

Proving Claim 12 is non trivial. The core of the proof consists in showing that any node that proposes a value v' other than the decided value v_d must find itself, at T_d, in a much earlier round than the earliest round occupied by any good node. In fact, we show something stronger: we choose a priority threshold large enough that any node, whether good or defective, that at T_d or later is within earshot of a good node (*i.e.*, whose message m can be collected by a good node), not only proposes v_d, but it does so with a *uCounter* large enough that allows whoever collects m to propose v_d with priority at least 1.

To see why those who propose v' are so far behind good nodes, note that the good node p_d that decided v_d at T_d must have received only messages proposing v_d for a long sequence of rounds, so long as to push v_d's priority over the $(6\mathcal{T} + 4)$ threshold required for a decision. Let's zoom in on that sequence of rounds. It took $6\mathcal{T}$ unanimous rounds for v_d to reach priority 1 (see Algorithm 1, line 20); after clearing that first hurdle, v_d's priority increased by 1 every $\mathcal{T}$ rounds.

Consider now the set S of messages collected by p_d during the long climb that took v_d's priority from 1 to $(6\mathcal{T} + 4)$. Any node p' that during this climb proposes something other than v_d faces a dilemma. It can either refuse to collect any message in S – but if it does so, it will advance more slowly than good nodes, and, by the time v_d's priority reaches the decision threshold, it will be so far behind that no good node will collect its messages. Or p' can try to keep up by collecting messages from S – but, if it wants to keep proposing $v' \neq v_d$, it can do so in at most one round during the entire climb: since the first message collected from S would reset v' priority to 0, any further message from S collected by p' in later rounds would have higher priority than the one of v', forcing p' to henceforth propose v_d instead of v'.

In short, since p' can collect messages from S in at most one round, to ensure that any node that in round r_d is within earshot of good nodes will propose v_d it suffices to choose a large enough priority threshold for deciding. In particular, setting the threshold to $(6\mathcal{T} + 4)$ ensures that (i) all messages collected by good nodes for round $(r_d - 1)$ will propose v_d, and (ii) v_d's *uCounter* in all these messages is at least $(6\mathcal{T} - 1)$, ensuring that all good nodes in round r_d will propose v_d with *uCounter* at least $6\mathcal{T}$, *i.e.*, with priority at least 1.

Finally, a simple induction argument shows that, if all good nodes propose v_d with priority at least 1 from r_d on, then any node that, from step $(T_d + 1)$ on, continues to propose a value other than v_d, will fall ever more behind good nodes, as it will be allowed to collect messages from good nodes only once every $6\mathcal{T}$ rounds, on pain of being forced to switch its proposed value to v_d.

5.3 Termination

The Termination property requires good nodes that stay active to eventually decide. Sandglass's Termination guarantee is probabilistic: for Termination to hold, Sandglass needs to be lucky, so that it can build a sequence of consecutive rounds during which all messages collected by good nodes propose the same value; long enough that the value will reach the priority required for a node to decide. Luck is required because Sandglass allows some randomness in the values that a node proposes: nodes are required to propose the highest priority value from any message collected in the previous round, but, if they receive multiple values with the same priority, they can choose among them uniformly at random.

To help us prove that luck befalls Sandglass with probability 1, we introduce the interdependent notions of *lucky period*, *lucky value*, and *lucky round*. Intuitively, a lucky period is a sequence of steps that leads to a decision: all nodes that are active in the step that immediately follows the end of the lucky period are guaranteed to decide in that step, if not earlier. A lucky round is simply the first round of a lucky period. What is more interesting is the quality that makes a period lucky: during a lucky period, whenever Sandglass allows nodes to use randomness in picking which value they will propose in the current round, they select the same value – the *lucky value* for that round.

A minimum requirement for a round's lucky value is that it should be a *plausible* value on which good nodes may converge, in the sense that it should not explicitly go counter the value that some good node is required to propose in that round. Concretely, if the messages collected by a good node require it to propose v and all other nodes can randomly choose between v and $\overline{v}$, then the round's lucky value better not be $\overline{v}$. In addition, to encourage the possibility of a lucky period, the lucky value should be *sticky*: we would like random choices to consistently pick the same value, round after round, unless doing so would make the value implausible.

In the end, Sandglass adopts a definition of lucky value (see [25]) that, in addition to upholding plausibility, has two additional properties that express its stickiness.

▶ **Property 13.** *In every round good nodes collect at least one message that proposes the lucky value of the previous round.*

▶ **Property 14.** *For the lucky value in the current round to change, some good node must have collected a different value with priority at least 1 from the previous round.*

Property 13 guarantees that under no circumstances the previous round's lucky value will simply be forgotten when moving to a new round; building on Property 13, Property 14 establishes that lucky values don't flip easily.

To prove that Sandglass guarantees Termination with probability 1, we then proceed in two steps.

First, we show that the *uCounter* of all good nodes active in the step that immediately follows the end of the lucky period reaches a value that allows these good nodes to decide. To this end, we begin by proving that, in any lucky period, the lucky value after a while becomes *locked*: specifically, we show that the lucky value v_ℓ at round $6\mathcal{T}$ in the lucky period remains the lucky value until the end of the lucky period, and, further, that after that round all good nodes propose v_ℓ. Then, leveraging techniques similar to those used to prove Agreement, we show that any node p' that proposes a value v' other that v_ℓ must fall behind good nodes during the lucky period. The reason is that, once v_ℓ is locked, p' can collect a message from a good node only every $6\mathcal{T}$ rounds. If it did it more often, p' would collect a message proposing v_ℓ from a good node while v' has priority 0, which would force p' to change its proposal to v_ℓ – even if v' and v_ℓ both had priority 0, and p' could choose randomly among them, it would have to propose v_ℓ in the next round, since v_ℓ is the lucky value. Thus, by choosing a sufficiently long lucky period, we ensure that nodes that propose values other that v_ℓ fall so far behind good nodes that v_ℓ's priority, for any good node that is active in the step right after the end of a lucky period, reaches the threshold necessary for deciding.

Second, we show that lucky periods occur with non-zero probability, since the probability of a certain outcome of random choices for a finite number of nodes during a finite number of steps is non-zero. Since in any infinite execution lucky periods appear infinitely often, it follows that any good node that stays active, no matter when it joins, is guaranteed to eventually decide.

6 Conclusion

Sandglass shows, for the first time, that it is possible to obtain consensus with deterministic safety in a permissionless model. This result suggests that it is the probabilistic nature of its PoW mechanism, rather that its permissionless model, that prevents deterministic safety in Nakamoto's consensus. It also opens up several additional interesting questions. First among them is to understand how the interplay between permissionlessness and the hybrid synchronous model shape the boundaries of what is possible, and at what cost. As we noted, Sandglass matches the $(n/2)$ bound of a benign model in an asynchronous network, even though a majority of its nodes are synchronously connected. Perhaps at the root of this result is that in both an asynchronous model and a permissionless hybrid one it is impossible for a node to know when it has received all the messages that were intended for it. Regardless, whether there exists a protocol that achieves deterministic safety and termination in a hybrid synchronous model remains an open question. Another natural question is whether there exists a deterministic solution to consensus in a hybrid-synchronous model with Byzantine failures. Answering these questions might pave the way to a qualitative improvement of permissionless systems that would provide deterministic guarantees; or, at the very least, give us more insight about the nature of consensus.

References

1 Ittai Abraham, Dahlia Malkhi, et al. The blockchain consensus layer and bft. *Bulletin of EATCS*, 3(123), 2017.

2 James Aspnes, Gauri Shah, and Jatin Shah. Wait-free consensus with infinite arrivals. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 524–533, 2002.

3 Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 27–30. ACM, 1983.

4 Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

5 Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros Praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 66–98. Springer, 2018.

6 Amir Dembo, Sreeram Kannan, Ertem Nusret Tas, David Tse, Pramod Viswanath, Xuechao Wang, and Ofer Zeitouni. Everything is a race and Nakamoto always wins. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 859–878, 2020.

7 Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

8 Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. Technical report, Massachusetts Inst. of Tech., Cambridge Lab for Computer Science, 1982.

9 Eli Gafni, Michael Merritt, and Gadi Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 161–169, 2001.

10 Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The Bitcoin backbone protocol: Analysis and applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 281–310. Springer, 2015.

11 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*, pages 51–68, 2017.

12 Maurice Herlihy. Blockchains and the future of distributed computing. In *Proceedings of the 2017 ACM Symposium on Principles of Distributed Computing (PODC '17)*, page 155, August 2017. Keynote Address.

13 Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is DAG. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 165–175, 2021.

14 Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual international cryptology conference*, pages 357–388. Springer, 2017.

15 Lucianna Kiffer, Rajmohan Rajaraman, and Abhi Shelat. A better method to analyze blockchain consistency. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 729–744, 2018.

16 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative Byzantine fault tolerance. *Communications of the ACM*, 51(11):86–95, November 2008.

17 Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

18 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

19 Andrew Lewis-Pye and Tim Roughgarden. Byzantine generals in the permissionless setting. *arXiv preprint*, 2021. arXiv:2101.07095.

20 Atsuki Momose and Ling Ren. Constant latency in sleepy consensus. *Cryptology ePrint Archive*, 2022.

21 Tal Moran and Ilan Orlov. Simple proofs of space-time and rational proofs of storage. In *Annual International Cryptology Conference*, pages 381–409. Springer, 2019.

22 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, December 2008. Accessed: 2015-07-01. URL: https://bitcoin.org/bitcoin.pdf.

23 Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 643–673. Springer, 2017.

24 Rafael Pass and Elaine Shi. The sleepy model of consensus. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 380–409. Springer, 2017.

25 Youer Pu, Lorenzo Alvisi, and Ittay Eyal. Safe permissionless consensus. Cryptology ePrint Archive, Paper 2022/796, 2022. URL: https://eprint.iacr.org/2022/796.

26 Michael O Rabin. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 403–409. IEEE, 1983.

27 Research and Markets. Blockchain market with covid-19 impact analysis, by component (platforms and services), provider (application, middleware, and infrastructure), type (private, public, and hybrid), organization size, application area, and region - global forecast to 2026. https://www.researchandmarkets.com, November 2021.

28 Yee Jiun Song and Robbert van Renesse. Bosco: One-step Byzantine asynchronous consensus. In *International Symposium on Distributed Computing*, pages 438–450. Springer, 2008.

29 TK Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.

30 Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

31 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

Packet Forwarding with a Locally Bursty Adversary

Will Rosenbaum
Amherst College, MA, USA

Abstract

We consider packet forwarding in the adversarial queueing theory (AQT) model introduced by Borodin et al. We introduce a refinement of the AQT (ρ, σ)-bounded adversary, which we call a *locally bursty adversary* (LBA) that parameterizes injection patterns jointly by edge utilization and packet origin. For constant $(O(1))$ parameters, the LBA model is strictly more permissive than the (ρ, σ) model. For example, there are injection patterns in the LBA model with constant parameters that can only be realized as (ρ, σ)-bounded injection patterns with $\rho + \sigma = \Omega(n)$ (where n is the network size). We show that the LBA model (unlike the (ρ, σ) model) is closed under packet bundling and discretization operations. Thus, the LBA model allows one to reduce the study of general (uniform) capacity networks and inhomogenous packet sizes to unit capacity networks with homogeneous packets.

On the algorithmic side, we focus on information gathering networks – i.e., networks in which all packets share a common destination, and the union of packet routes forms a tree. We show that the Odd-Even Downhill (OED) forwarding protocol described independently by Dobrev et al. and Patt-Shamir and Rosenbaum achieves buffer space usage of $O(\log n)$ against all LBAs with constant parameters. OED is a local protocol, but we show that the upper bound is tight even when compared to centralized protocols. Our lower bound for the LBA model is in contrast to the (ρ, σ)-model, where centralized protocols can achieve worst-case buffer space usage $O(1)$ for $\rho, \sigma = O(1)$, while the $O(\log n)$ upper bound for OED is optimal only for local protocols.

2012 ACM Subject Classification Theory of computation → Routing and network design problems; Networks → Packet scheduling; Theory of computation → Distributed algorithms; Theory of computation → Distributed computing models

Keywords and phrases packet forwarding, packet scheduling, adversarial queueing theory, network calculus, odd-even downhill forwarding, locally bursty adversary, local algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.34

Acknowledgements This work was born from numerous discussions with Boaz Patt-Shamir, to whom I am eternally grateful. I thank the anonymous reviewers for their thoughtful commentary which helped improve this paper.

1 Introduction

Routing and forwarding are fundamental operations in the study of networks. In this context, commodities – for example, data packets, fluid flows, or physical objects – appear at various places in a network, and must be transferred to prescribed destinations. Movement is restricted by the network's topology. The goal is to get the commodities from source to destination as efficiently as possible. *Routing* is the process of determining routes for the commodities to follow from source to destination, while *forwarding* determines the particular schedule by which items – which we will henceforth refer to as *packets* – move in the network. In this work, we focus on the process of forwarding packets, assuming their routes are pre-determined.

Two well-studied models of packet forwarding in networks are the adversarial queueing theory (AQT) model introduced by Borodin et al. [2] and the network calculus model introduced by Cruz [4, 5]. In both models, packets are assumed to have prescribed routes from source to destination. Both models also parameterize packet arrivals in terms of long-

© Will Rosenbaum;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 34; pp. 34:1–34:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

term average rates and short-term burstiness in order to disallow trivially infeasible injection patterns that exceed network capacity constraints. AQT and network calculus also differ in some crucial ways. AQT examines injections of discrete, indivisible packets at discrete time intervals, and forwarding occurs in synchronous rounds. In network calculus, on the other hand, packets are modeled as continuous flows and forwarding is a continuous-time processes. Nonetheless, these flows can be discretized (or "packetized") to be processed discretely. One of the goals of this paper is to draw tighter connections between analogous parameters in the AQT and network calculus models under the process of discretization.

In both AQT and network calculus, one natural measure of efficiency is the buffer space usage of nodes in the network. That is, how much memory is required at each buffer in order to store packets that are *en route* to their destinations. Traditionally, AQT has focused on a qualitative measure of space usage, called *stability*, which merely requires that the space usage of a protocol remains bounded (by some function of the network parameters) for all time. A notable early exception is the work of Adler and Rosén [1], which gives a quantitative buffer space upper bound for longest-in-system scheduling when the network is a directed acyclic graph.

AQT has also traditionally focused on *greedy* forwarding protocols – i.e., protocols for which every non-empty buffer forwards as many packets as it can in each round (subject to capacity constraints). A more recent series of work [8, 6, 10, 11, 9] initiated by Miller and Patt-Shamir [8] studies quantitative buffer space bounds for non-greedy forwarding policies. In particular, these works show that in restricted network topologies (single-destination paths and trees), non-greedy forwarding protocols can achieve significantly better buffer space usage than greedy protocols. Specifically, non-greedy centralized forwarding protocols can achieve $O(1)$ buffer space usage [8, 9], while $\Theta(\log n)$ buffer space is necessary and sufficient for local (distributed) protocols [6, 10] (where n is the number of buffers in the network). The work of Patt-Shamir and Rosenbaum [11] shows there is a smooth trade-off between a protocol's *locality* and optimal buffer space usage: if each node determines how may packets to forward based on the state of its distance d neighborhood, then $\Theta(\frac{1}{d}\log n)$ buffer space is necessary and sufficient. These bounds are in contrast to greedy protocols, which require $\Omega(n)$ buffer space in the worst case.

The bounds described in the preceding paragraph refer to the AQT injection model in which edges in the network have uniform unit capacities – only one packet may cross any edge in a given round – and the average injection rate ρ satisfies $\rho \leq 1$, and the burst parameter σ satisfies $\sigma = O(1)$ (cf. Definition 2.1). The algorithms can be generalized to general uniform edge capacities ($C \geq 1$ packets can cross each edge in a round), but the generalized algorithms are both more cumbersome to express, and correspondingly subtle to reason about (see, e.g., Section 1.1 in [6]). When dealing with general capacities, discretizations of continuous flows, and heterogeneous (indivisible) packets, a natural strategy is to bundle packets into "jumbo packets" [12]. This procedure can, however, lead to large bursts in the appearance of jumbo packets in the network, even if the injection process has a small burst parameter (see e.g., Remark 2.10). Thus applying the *analysis* of the relatively simple unit-capacity versions of algorithms in [8, 6, 10, 11, 9] directly to jumbo packets may not show any improvement over greedy algorithms.

1.1 Our Contributions

In this paper, we introduce a refinement of Borodin et al. [2]'s (ρ, σ) parameterization of injection patterns, that we call a the ***local burst model*** (see Definition 2.2). In addition to the asymptotic rate ρ and *global* burst parameter σ, the local burst model has a third

parameter, β – the *local* burst parameter, that accounts for simultaneous bursts occurring at distinct injection sites. Thus, for small values of β, a locally (ρ, σ, β)-bounded injection pattern may still allow for large (e.g, $\Omega(n)$) simultaneous packet injections, so long as not too many packets are injected into the same buffer.

We prove that locally bursty injection patterns are essentially characterized as discretizations of what we call "locally dependent flows" (Definition 2.6) with similar parameters – see Lemmas 2.7 and 2.9. We use this characterization to show that applying packet bundling to a locally bursty injection pattern yields another locally bursty adversary with similar parameters – see Proposition 3.1. Consequently, any space efficient algorithm for unit capacity networks and homogeneous unit sized packets can be applied as a black-box to bundled jumbo packets to achieve similar buffer space usage to the unit capacity case (Corollary 3.2). We then show that a small modification of the framework can also be applied to the setting of heterogeneous packet sizes.

On the algorithmic side, we analyze the *odd-even downhill* (OED) forwarding protocol of [6, 10] against locally bursty adversaries. We show that for constant ($O(1)$) parameters ρ, σ, and β, OED achieves worst-case buffer space $O(\log n)$ in information gathering networks of size n – see Theorem 4.2. This result is strictly stronger than the analyses of [6, 10], as there are locally bursty injection patterns with $\beta = O(1)$ that can only be realized in the classical (ρ, σ) model for $\sigma = \Omega(n)$. Combining our analysis of the OED protocol together with flow discretization and/or bundling, buffer space of $O(\log n)$ can be achieved for forwarding with general capacities, heterogeneous packets, and continuous flows – see Section 4.1.

Finally, in Section 5, we prove a matching lower bound of $\Omega(\log n)$ for any *centralized randomized* protocol against locally bursty adversaries (Theorems 5.1 and A.1). This lower bound is in contrast to the deterministic upper bounds of [8, 11, 9] which show that $O(1)$ buffer space is achievable for (ρ, σ)-bounded adversaries for centralized and "semi-local" protocols. Thus, our lower bound shows that the local burst model (with constant parameters) gives the adversary strictly more power to inflict large ($\omega(1)$) buffer space usage on *centralized* algorithms. The performance of the asymptotically optimal *local* protocol is the same for the local burst and traditional injection models, and in the local burst model, the (local) OED protocol is asymptotically optimal, even when compared to centralized protocols.

1.2 Discussion of Our Results

The most natural application domains for this work are networks consisting of tightly synchronized nodes, such as network-on-chips (NoCs) [7], software defined networks (SDNs) [13], and sensor networks. In these contexts, trees and grids are common network topologies. (In the case of grids, "single bend" routing allows one to treat the network essentially as a disjoint union of paths.) Thus, while the topologies we consider are highly restricted, the family of topologies is a fundamental and frequently used family for applications in which our techniques might be applied.

In NoCs, SDNs, and sensor networks, the rate and source of packet injections may be highly variable. Thus, the parameterizations of packet injections in both the standard AQT model and the network calculus model may be too coarse to model the actual buffer space requirement of observed injection patterns effectively. In the AQT model, allowing for multiple simultaneous packet injections into different buffers requires a large burst parameter, σ, even though the resulting injection pattern may be handled using buffer space $\ll \sigma$ (see Example 2.4). On the other hand, the traditional network calculus model does not account for dependencies between rates of packet injections into different buffers over time. Thus, a standard analysis may severely overestimate the bandwidth or buffer space required to

handle a given injection pattern. Our locally bursty injection model (and its continuous analogue described in Section 2.2) refine both the AQT and network calculus models so as to give more precise bounds on the buffer space requirement of many natural packet injection patterns.

Together with the upper bounds of [8, 11, 9], Theorems 5.1 and A.1 imply that the locally bursty injection model (with constant parameters) gives an adversary strictly greater power to inflict large buffer space usage against *centralized* and semi-local forwarding protocols. However, Theorem 4.2 implies that the *local* OED forwarding protocol achieves asymptotically optimal buffer space usage. Thus, for locally bursty injection patterns, there is no (asymptotic) advantage to implementing a centralized protocol, while OED still gives an exponential improvement over greedy protocols. We believe this insight may be valuable in VLSI design where protocols like OED could be implemented at a hardware level in order to reduce buffer space requirements. Hardware implementations of similar protocols have been proposed, for example in [3], in order to achieve decentralized (gradient) clock synchronization.

2 Model and Preliminaries

We model a packet forwarding network as a directed graph, $G = (V, E)$. Each edge $e = (u, v) \in E$ has an associated buffer that stores packets in node u as they wait to cross the edge (u, v) to v. We use the notation $e = (u, v)$ to denote both the edge in G and its associated buffer.

In our model, an execution proceeds in synchronous rounds. Each round consists of two steps: an ***injection step*** in which new packets arrive in the network, and a ***forwarding step*** in which buffers forward packets across edges of the graph. During the forwarding step, each buffer chooses a subset of packets to forward, and forwards those packets across the edge (u, v) associated with the buffer. These packets arrive at their next location – either another buffer in node u, or are delivered to their destination – before the beginning of the next round. Each edge e, has a ***capacity*** $C(e)$, which is the maximum number of packets that can cross e in a single forwarding step.

At a given time t, we use $L^t(e)$ to denote the contents of buffer e during round t between the injection and forwarding steps. $|L^t(e)|$ is the ***load*** of e – i.e., number of packets stored in buffer e.

A ***packet*** p is a pair (t, P) where $t \in \mathbf{N}$ and $P = (v_0, v_1, \ldots, v_\ell)$ is a directed path in G. The interpretation is that t indicates the time (round) at which P is injected, and P specifies a ***route*** from P's ***source***, $e_0 = (v_0, v_1)$ to P's ***destination*** v_ℓ. An ***adversary*** or ***injection pattern*** A is a multi-set of packets.

Given a packet $p = (t, (v_0, v_1, \ldots, v_\ell))$, we say that p's route ***contains*** an edge $e \in E$ if $e = (v_i, v_{i+1})$ for some $i \in [\ell - 1]$. For a fixed adversary A and time interval $T = [r, s] \subseteq \mathbf{N}$, we define $N^T(e)$ to be the number of packets injected during times $t \in T$ whose routes contain e. That is

$$N^T(e) = |\{(t, P) \in A \mid t \in T \text{ and } P \text{ contains } e\}| .$$

We also define a more refined measure of utilization of an edge e that differentiates packets according to their origins. Specifically, for any subset $S \subseteq E$, we define

$$N_S^T(e) = |\{(t, P) \in A \mid t \in T, (v_0, v_1) \in S, \text{ and } P \text{ contains } e\}|$$

In particular, we have $N^T(e) = N_E^T(e)$. In the adversarial queueing model (AQT) of Borodin et al. [2], the edge utilization of an adversary A is parameterized is follows.

▶ **Definition 2.1.** *Given $\rho > 0$ and $\sigma \geq 0$, we say that an adversary A is* (ρ, σ)***-bounded*** *if for all e and (finite) intervals $T \subseteq \mathbf{N}$ we have*

$$N^T(e) \leq \rho\,|T| + \sigma. \tag{1}$$

We denote the family of (ρ, σ)-bounded adversaries by $\mathcal{A}(\rho, \sigma)$.

For a (ρ, σ)-bounded adversary, the parameter ρ is an upper bound on the maximum average utilization of an edge in the network, while σ measures "burstiness" – the amount by which the average can be exceeded over any time interval. For example, taking T with $|T| = 1$, (1) implies that at most $\rho + \sigma$ packets are injected into any buffer in any single round.

2.1 Locally Bursty Adversaries

Here, we define a more refined parameterization of adversaries, which we call the ***local burst model***. We refer to adversaries parameterized by the local burst model as ***locally bursty adversaries***, or ***LBA***s.

▶ **Definition 2.2.** *Let A be an adversary, $\rho > 0$, $\sigma \geq 0$ and $\beta : E \to \mathbf{N}$. Then we say that A is* ***locally*** (ρ, σ, β)***-bounded*** *if for all finite intervals $T \subseteq \mathbf{N}$, subsets $S \subseteq E$ and $e \in E$, we have*

$$N_S^T(e) \leq \rho\,|T| + \sigma + \sum_{f \in S} \beta(f). \tag{2}$$

That is, for every subset S of buffers, the rate of injections into S that cross e only ever exceeds ρ by σ more than the sum of the $\beta(e)$ for $e \in S$. We denote the family of local (ρ, σ, β)-bounded adversaries by $\mathcal{L}(\rho, \sigma, \beta)$. In the case that $A \in \mathcal{L}(\rho, \sigma, \beta)$ and there is a constant B such that $\beta(e) \leq B$ for all buffers e, we will say that A is local (ρ, σ, B)-bounded.

We formalize the following observation that gives a relationship between the parameters of (ρ, σ)-bounded adversaries and local (ρ, σ, β)-bounded adversaries.

▶ **Observation 2.3.** *Fix a network G and parameters ρ, σ, and $\beta : E \to \mathbf{N}$. Suppose $A \in \mathcal{L}(\rho, \sigma, \beta)$. Then $A \in \mathcal{A}(\rho, \sigma')$ for $\sigma' = \sigma + \sum_{e \in E} \beta(e)$.*

▶ **Example 2.4.** Let G be the ***single-destination path*** of size n. That is, $G = (V, E)$ where $V = \{1, 2, \ldots, n, n+1\}$ and $E = \{(i, i+1) \mid i \in [n]\}$. Further, all injected packets have destination $n + 1$. We consider two injection patterns, A_0 and A_1

A_0*:* in rounds $1, n+1, 2n+1, \ldots, kn+1, \ldots$, there are n packets injected into buffer 1 with destination $n + 1$.

A_1*:* in rounds $1, n+1, 2n+1, \ldots, kn+1, \ldots$, one packet is injected into each buffer $i = 1, 2, \ldots, n$ with destination $n + 1$.

Observe that both adversaries are in $\mathcal{A}(1, n-1)$, but not in $\mathcal{A}(1, \sigma)$ for any $\sigma < n - 1$. Thus, the parameters of Definition 2.1 do not distinguish A_0 and A_1. Yet A_0 and A_1 have vastly different buffer space requirements. A_0 requires buffer 1 to have space n for any forwarding protocol, while simple greedy forwarding for A_1 will achieve buffer space usage $|L^t(i)| \leq 1$ for all t and i.

The parameters of the local burst model, however, can distinguish between A_0 and A_1. $A_1 \in \mathcal{L}(1, 0, 1)$ (i.e., $\beta(i) = 1$ for all i), while $A_0 \in \mathcal{L}(1, \sigma, \beta)$ only for $\sigma + \beta(1) \geq n - 1$. We will show that in the case of *information gathering networks* – networks in which all

packets share a common destination and the union of their routes forms a tree – all local $\mathcal{L}(1, \sigma, B)$-bounded adversaries can be forwarded using $O(B \log n + \sigma)$ space. Thus, the *local* burst parameter β gives a more refined understanding of the buffer space requirement of a given injection pattern.

2.2 Flows

Another well-studied model for packet forwarding is the network calculus model introduced by Cruz [4, 5]. In the network calculus, packets are associated with *flows*, and their arrivals are modeled as continuous time processes.

▶ **Definition 2.5.** *Given a network $G = (V, E)$, A* ***flow*** *$\phi = (a, P)$ consists of a right-continuous arrival curve $a : \mathbf{R} \to \mathbf{R}$ and associated path P. We say that ϕ has* ***rate*** *(at most) r and burst parameter b if for all $s < t$, the arrival curve a satisfies*

$$a(t) - a(s) \leq r \cdot (t - s) + b. \tag{3}$$

By convention, we assume $a(t) = 0$ for all $t < 0$.

For a single flow ϕ, the parameters r and b are analogous to the rate and burst parameters ρ and σ in Definition 2.1. However, in a flow ϕ, all packets share a common route, P. In particular, all packets associated with ϕ are injected to the same buffer and have the same destination.

In order to consider scenarios in which packets have multiple routes, we must consider multiple concurrent flows. In this setting, the analogy between equations (1) and (3) breaks down, as the former bounds the total number packets utilizing any particular edge, while the latter bounds the arrivals of packets in flows (i.e., along entire paths, rather than individual edges). In order to tighten the connection between the AQT injection model and flows, we introduce a ***dependent*** flow model in which we constrain the sum of arrival rates of flows across edge.

▶ **Definition 2.6.** *Let $G = (V, E)$ be a network and $\Phi = \{\phi\}$ be a family of flows. For an edge $e \in E$, let Φ_e denote the set of flows in Φ whose paths contain e. That is,*

$$\Phi_e = \{(a, P) \in \Phi \mid e \in P\}.$$

Suppose each $\phi \in \Phi$ obeys a r_ϕ, b_ϕ bound as in (3). We say that Φ obeys a ***locally dependent rate bound*** *r and* ***global burst parameter*** *σ if for every edge e, every set $\Psi \subseteq \Phi_e$ of flows, and all times s, t, we have*

$$\sum_{\phi \in \Psi} (a_\phi(t) - a_\phi(s)) \leq r \cdot (t - s) + \sigma + \sum_{\phi \in \Psi} b_\phi. \tag{4}$$

We note the similarity between equations (4) and (2). In fact, Definition 2.6 is a strict generalization of the LBA model: Given any injection pattern A, we can associate a family Φ_A of flows with A. Specifically, we define Φ_A to be

$$\Phi_A = \left\{ (a, P) \;\middle|\; (t, P) \in A \text{ and } a(t) = \sum_{s \in \mathbf{N},\ s \leq t} |\{(s, P) \in A\}| \right\} \tag{5}$$

With this association, the following lemma is clear.

▶ **Lemma 2.7.** *Suppose A is a locally (ρ, σ, β)-bounded adversary, and let Φ_A be the corresponding flow defined by (5). Then for each flow $\phi = (a_\phi, P_\phi) \in \Phi_A$, a has rate at most ρ, global burst parameter σ, and local burst parameter $b_\phi = \beta(\mathrm{ini}_\phi)$, where ini_ϕ denotes the initial buffer in ϕ's path. Moreover, Φ_A obeys a locally dependent rate bound of ρ.*

Conversely, LBAs arise naturally as discretizations (packetizations) of (locally dependent) flows. We formalize this connection in the following definition and lemma.

▶ **Definition 2.8.** *Let $G = (V, E)$ be a network and Φ a family of flows on G. The **discretization** of Φ is the AQT injection pattern A_Φ defined as follows. For each flow $\phi = (a_\phi, P_\phi) \in \Phi$ and time $t = \mathbf{N}$, A_Φ contains $\lfloor a_\phi(t)\rfloor - \lfloor a_\phi(t-1)\rfloor$ packets injected at time t with route P_ϕ.*

We can view the discretization of a flow as forming packets via the following process. Each buffer maintains a set of (complete) packets, as well as a reserve of "fractional" packets associated with each flow originating at the buffer. At times $s \in (t-1, t]$, flows enter a buffer e. At time t, the integral parts of each flow that has not yet been bundled as packets are injected as complete packets into the buffer, while the fractional remainder is reserved. The following lemma shows that for flows obeying a locally dependent rate bound, the resulting packet injection pattern is locally bounded as well.

▶ **Lemma 2.9.** *Let $G = (V, E)$ be a graph and Φ a family of flows on G. For each $\phi \in \Phi$, let ini_ϕ denote the initial buffer in ϕ's path. Suppose Φ obeys a locally dependent rate bound of r with global burst parameter σ, and define the function $\beta : E \to \mathbf{N}$ by*

$$\beta(e) = \sum_{\phi\,:\,\mathrm{ini}_\phi = e} (1 + b_\phi).$$

Then the discretization A_Φ of Φ is locally (r, σ, β) bounded.

Proof. Fix a set $S \subseteq E$ of initial buffers, an edge e, and (discrete) time interval $T = [t_0, t_1]$. Let $A = A_\Phi$, and let $\Psi \subseteq \Phi_e$ be the subset of flows containing e and with origin in S. We compute

$$\begin{aligned}
N_S^T(e) &= |\{(t, P) \in A \mid t \in T, (v_0, v_1) \in S, \text{ and } P \text{ contains } e\}| \\
&= \sum_{\phi\in\Psi}\sum_{s\in T} (\lfloor a_\phi(s)\rfloor - \lfloor a_\phi(s-1)\rfloor) \\
&= \sum_{\phi\in\Psi} (\lfloor a_\phi(t_1)\rfloor - \lfloor a_\phi(t_0-1)\rfloor) \\
&\leq r \cdot (t_1 - t_0 + 1) + \sigma + \sum_{\psi\in\Phi} (1 + b_\phi) \qquad (6) \\
&= r \cdot |T| + \sigma + \sum_{e\in S} \beta(e).
\end{aligned}$$

In Equation (6), we use the fact that $\lfloor a\rfloor - \lfloor b\rfloor \leq 1 + a - b$. ◀

▶ Remark 2.10. The result of Lemma 2.9 is a significant refinement of the analogous statement for the standard (ρ, σ) burst model. To see this, consider the single destination path (Example 2.4), and take $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$ to be the family of flows where each ϕ_i has arrival curve $a(t) = \frac{1}{n}t$ and associated path $P_i = (i, i+1, \ldots, n+1)$. In the associated injection pattern A_Φ, one packet is injected into every buffer at times $n, 2n, 3n, \ldots$ (cf. A_1 in Example 2.4). Even though flows in Φ have burst parameter 0, large bursts appear in A_Φ as the result of the rounding process. Nonetheless, Lemma 2.9 asserts that A_ϕ is *locally* $(1, 0, 1)$-bounded, while the injection pattern is only $(1, \sigma)$-bounded for $\sigma \geq n - 1$.

3 Packet Bundling

In this section we assume that for a network $G = (V, E)$, all edges have the same (integral) capacity C. We examine the following strategy for dealing with general uniform capacity networks: when packets arrive in a buffer, they are set in a reserve buffer until sufficiently many (e.g., C) packets occupy the reserve buffer. Then the packets are bundled together, and treated as one indivisible "jumbo" packet. This process is appealing because if all jumbo packets have size C, then forwarding protocols designed for unit capacities can be applied to jumbo packets. Thus, the approach sidesteps potential subtleties in reasoning about general capacities (see Section 1.1 in [6]).

Our main results in this section show that if the original packet injection pattern obeys an LBA bound, then the resulting injection pattern of jumbo packets obeys a similar LBA bound with the parameters scaled down. Thus, if any algorithm guarantees some buffer space usage for unit capacity networks, then applying the same algorithm to jumbo packets will automatically give an analogous bound for general capacities.

3.1 Uniform Packets

We first consider the case where all packets have unit size (as in the standard AQT model), but all edges in the network have capacity $C \geq 1$. Now let A be any locally (C, σ, β)-bounded adversary, and let $\Phi = \Phi_A$ be the corresponding family of flows. We define the C***-reduction*** of Φ, denoted $\frac{1}{C}\Phi$, to be

$$\frac{1}{C}\Phi = \left\{ \left(\frac{1}{C}a, P \right) \,\middle|\, (a, P) \in \Phi \right\}.$$

Similarly, we define the C-reduction of A, denoted $\frac{1}{C}A$, to be the discretization (Definition 2.8) of $\frac{1}{C}\Phi$.

Observe that $\frac{1}{C}A$ is derived from A via precisely the process of forming jumbo packets as described above. The following proposition follows immediately from Lemmas 2.7 and 2.9.

▶ **Proposition 3.1.** *Suppose A is locally (ρ, σ, β)-bounded. Then, $\frac{1}{C}A$ is locally $(\rho/C, \sigma/C, 1+\beta/C)$-bounded.*

Again, we emphasize that the analogue of Proposition 3.1 is not true for the standard (ρ, σ)-bounded adversary model. The proposition has the following consequence.

▶ **Corollary 3.2.** *Suppose F is a forwarding protocol that for any locally $(1, \sigma, \beta)$-bounded adversary A on a unit-capacity network $G = (V, E)$ achieves buffer space usage*

$$\sup_{e \in E, t \in \mathbf{N}} \left| L^t(e) \right| \leq f_G(\sigma, \beta).$$

Then for any uniform capacity C and locally (C, σ, β)-bounded adversary A, applying F to $\frac{1}{C}A$ achieves buffer space usage

$$\sup_{e \in E, t \in \mathbf{N}} \left| L^t(e) \right| \leq C f_G(\sigma/C, 1 + \beta/C) + C.$$

We note that the additive C term in the final expression comes from the need to store packets that have not yet been bundled.

3.2 Heterogeneous Packets

The framework described in Sections 2.2 and 3.1 shows how forwarding protocols for the AQT model with unit edge capacities can be applied to (1) discretizations of continuous flows, and (2) AQT adversaries with arbitrary uniform edge capacities and (uniform) unit-sized packets. Here, we describe a slight modification of the framework that allows for indivisible packets with heterogeneous sizes. To this end, we augment the AQT model as follows:

- Each packet p has an associated size, denoted $w(p)$.
- In a single round, an edge with capacity C can forward a set of packets whose sizes sum to at most C.
- An adversary A is locally (ρ, σ, β) bounded if for any subset S of buffers, any edge e, and in any T consecutive rounds, the sum of *sizes* of packets injected into S whose paths contain e is at most $\rho \cdot T + \sigma + \sum_{f \in S} \beta(f)$.

The following example shows one complication caused by indivisible heterogeneous packets.

▶ **Example 3.3.** Consider a single edge e with capacity 1. Then a $(1, 1)$-bounded adversary can inject 3 packets of size $2/3$ every 2 rounds that must cross e. Since e has capacity 1, it can only forward a single packet each round. Thus, the injection pattern is infeasible (i.e., cannot be handled with finite buffer space).

We can preclude infeasible injection patterns (such as Example 3.3) by further restricting the allowable injection rate. Consider the following bundling procedure: when packets are injected into a buffer, they are placed in a reserve buffer. If the load of the reserve buffer exceeds $\frac{1}{2}C$, then its contents are bundled into packets, each of whose total load is at least $\frac{1}{2}C$. Arguing as before, if the original injection pattern is locally (ρ, σ, β)-bounded with $\rho \leq \frac{1}{2}C$, then the resulting injection pattern of bundled packets is locally $(1, \sigma/C, 1 + \beta/C)$-bounded. Note that even though the rate of the adversary is $\rho \leq 1/2$, the rate of the bundled injections can be as large as 1. This occurs, for example, if the adversary always injects $C/2$ packets into a single buffer each round. These packets are then bundled, resulting in one complete bundle appearing each round.

4 OED Upper Bound

In this section we prove worst-case buffer space upper bounds for "information gathering networks" with a locally bursty adversary – i.e., instances in which all packets share a common destination and the union of trajectories of all packets forms a tree. Specifically, we show that the odd-even downhill (ODE) algorithm of [6, 10] requires $O(B \log n + \sigma)$ buffer space for any (ρ, σ, β)-bounded adversary for which $\beta(e) \leq B$ for all buffers e. For clarity and notational simplicity we describe the algorithm and argument in the simpler setting where the network consists of a path. All of the results remain true for general information gathering networks, and analogous arguments follow using the terminology and preliminary results described in [10].

In the case of the single destination path, the network $G = (V, E)$ consists of a path: $V = \{1, 2, \ldots, n+1\}$, and $E = \{(i, i+1) \mid i \leq n\}$. All packets share the destination $n+1$, though they can be injected into any buffer. To cut down on notational clutter, we associate a buffer $(i, i+1)$ with its index i. In this setting, we describe the ***Odd-Even Downhill*** or ***OED*** algorithm independently introduced by Dobrev et al. [6] and Patt-Shamir and Rosenbaum [10]. Following Section 3, we assume that all edge capacities are 1, and that all packets have unit size. To simplify notation, we use $L^t(i)$ to denote the the *number* of packets in buffer i immediately before the forwarding step of round t.

▶ **Definition 4.1.** *The **OED rule** stipulates that i forwards a packet in round t if and only if one of the following conditions is satisfied:*
1. $L^t(i) > L^t(i+1)$*, or*
2. $L^t(i) = L^t(i+1)$ *and* $L^t(i)$ *is* odd.

By convention, we set $L^t(n+1) = 0$ *for all* t.

Both original papers [6, 10] show that for all (ρ, σ)-bounded adversaries, the maximum buffer load under OED forwarding is $O(\log n + \sigma)$. We will show that OED forwarding achieves similar buffer space usage for any local (ρ, σ, β)-bounded adversary.

▶ **Theorem 4.2.** *Let G be a single-destination path of size $n+1$, and let A be any local (ρ, σ, β)-bounded adversary where $\beta(i) \leq B$ for all i. Then the worst case buffer load is $O(B \log n + \sigma)$. That is,*

$$\sup_{t,i} L^t(i) = O(B \log n + \sigma).$$

Our proof of Theorem 4.2 follows the analysis of OED presented in [10]. Specifically, their analysis considers the evolution of *plateaus* in the network.

▶ **Definition 4.3** (cf. [10])**.** *Let G be a single destination path and $L^t : V \to \mathbf{N}$ a **configuration** – i.e., assignment of loads to buffers – at some time t. We say that an interval $I = [a, b] \subseteq V$ is a **plateau** of **height** h if I is a maximal sub-interval of V such that for all $i \in I$, $L^t(i) \geq h$. That is, every buffer in I has load at least h, and there is no larger interval I' containing I with this property. We say that I is an **even plateau** if I is a plateau of height h for some even number h.*

We think of packets in G as being arranged vertically in buffers – see Figure 1. Since packets share the same destination, for the purposes of our load analysis, we can treat all packets in a buffer as indistinguishable.[1] We refer to the ***height*** of a packet in a buffer as one greater than the number of packets below it. Thus, for a buffer with load 1, its sole packet is at height 1; a buffer with two packets has one at height 1 and the second at height 2, etc. We say that a packet P is ***above*** a plateau I of height h if $\mathrm{ht}(P) > h$. Given a configuration $L : V \to \mathbf{N}$ and a plateau I of height h, we denote the number of packets above I by $L_h(I)$. That is,

$$L_h(I) = \sum_{i \in I} (L(i) - h). \tag{7}$$

OED forwarding does not specify *which* packet is forwarded when a buffer is forwarded, and the maximum load analysis of the algorithm is independent of this choice. Nonetheless, for the purposes of bookkeeping, it will be convenient to adopt the following conventions:
1. Whenever a packet is injected or received as the result of forwarding, it occupies the highest position in its buffer;
2. When a buffer forwards a packet, the highest packet in the buffer is forwarded.

That is – for the purposes of bookkeeping – we assume that the buffers operate as LIFO (last-in, first-out) stacks.

With these conventions, we make some preliminary observations about the movement of packets in an execution of the OED algorithm.

[1] In order to analyze packet latency, one should distinguish packets by their age.

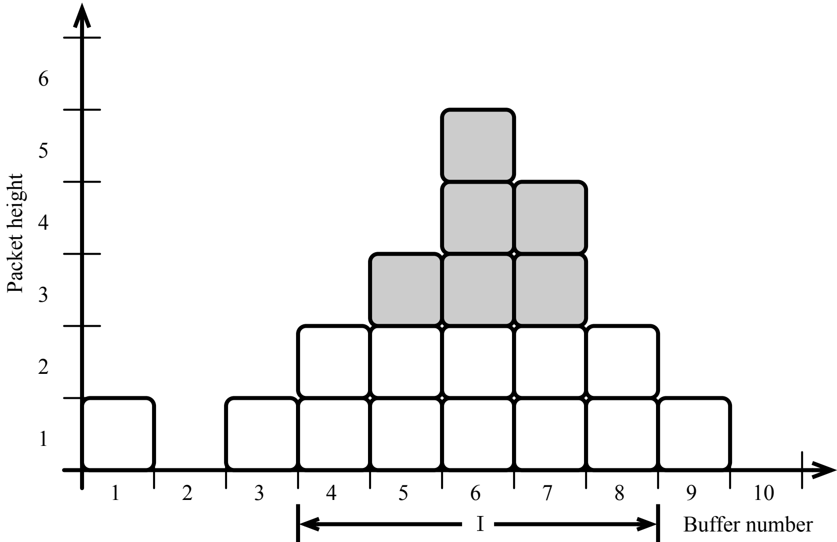


Figure 1 A configuration of packets. Each column represents a buffer, labeled 1 through 10, while the vertical axis indicates heights. The load of each buffer corresponds to the height of the highest packet in the buffer; for example, $L(6) = 5$. The indicated interval $I = [4, 8]$ is a plateau of height 2. The six shaded packets sit above the plateau I, so that $L_2(I) = 6$. Corollary 4.8 states that the number of packets above I is at most $(B + 1)\,|I| + \sigma$ for any locally $(1, \sigma, B)$-bounded adversary. The proof of Theorem 4.2 applies this corollary inductively to the nested sequence of even plateaus containing the buffer with maximum load to show that the maximum load is at most logarithmic in the size of the network.

▶ **Lemma 4.4.** *Suppose $L : V \to \mathbf{N}$ is a configuration immediately before forwarding and L' the configuration afterward. Suppose $I = [a, b] \subseteq V$ is an even plateau of height h in configuration L. Then in configuration L', for all $i \in [a, b-1]$, we have $L'(i) \geq h$. Thus, in L', the interval $[a, b-1]$ is contained in a plateau I' of height h.*

Proof. Suppose $i \in [a, b-1]$. Then in L, we have $L(i), L(i+1) \geq h$. Since h is even, i will not forward unless $L(i) \geq h+1$. Therefore, $L'(i) \geq h + 1 - 1 = h$. ◀

▶ **Lemma 4.5.** *For any packet P, let $\mathrm{ht}^t(P)$ denote the height of P at time t,*[2] *and let h be an even number. If $\mathrm{ht}^t(P) \leq h$, then for all $s \geq t$ we have $\mathrm{ht}^s(P) \leq h$.*

Proof. By our conventions of packet movement, the height of P in a fixed buffer is unchanged until it is forwarded. Since the top packet is always forwarded, P can only be forwarded if $\mathrm{ht}(P) = L(i)$, where i is the buffer containing P before forwarding. Since $\mathrm{ht}(P) \leq h$ we also have $L(i) \leq h$. Since h is even, i only forwards if $L(i+1) \leq h - 1$. Therefore, the height of P *after* forwarding is at most $L(i+1) + 1 \leq h - 1 + 1 = h$. Thus, P remains at height at most h. ◀

▶ **Corollary 4.6.** *Suppose $L : V \to \mathbf{N}$ is a configuration and $I = [a, b]$ is an even plateau of height h. Suppose a packet P was injected at time s, and at time $t \geq s$ sits above I (i.e., P occupies a buffer $j \in I$ and $\mathrm{ht}^t(P) > h$). Then P was injected into a buffer in I with an initial height $\mathrm{ht}^s(P) > h$.*

[2] Note that this quantity is well-defined for all t (until P is delivered to its destination) by the LIFO conventions for height and packet movement.

Proof. Suppose that at time t, P occupies buffer $j \in [a, b]$. Since packets are only forwarded to a buffer with larger index, P was injected at some buffer $i \leq j$ at some time $t_0 \leq t$. By Lemma 4.5, for all $s \in [t_0, t]$ we have $\mathrm{ht}^s(P) > h$, whence the second assertion of the corollary holds.

For the first assertion, we must show that $i \in I$. To this end, for each time $s \in [t_0, t]$, let $I_s = [a_s, b_s]$ denote the plateau of height h in which P is contained at time s. Thus, I_{t_0} is the plateau above which P is initially injected so that $j \in I_{t_0}$, and $I_t = I = [a, b]$. By Lemma 4.4, for all $s < t$, we have $[a_s, b_s - 1] \subseteq I_{s+1} = [a_{s+1}, b_{s+1}]$. Therefore, we have $a_s \geq a_{s+1}$ for all s. Thus, by induction, we have $a_t \leq a_{t_0} \leq j$, where the second inequality holds because $j \in I_{t_0}$. This gives the desired result. ◀

We now quote a lemma from [10], which bounds the number of packets above even plateaus for (ρ, σ)-bounded adversaries.

▶ **Lemma 4.7** (cf. Lemma 3.4 in [10]). *Let A be a (ρ, σ)-bounded adversary and suppose L is a configuration realized by OED forwarding. Suppose I is an even plateau of height h. Then*

$$L_h(I) \leq |I| + \sigma.$$

▶ **Corollary 4.8.** *Let A be a local (ρ, σ, β)-bounded adversary with $\beta(i) \leq B$ for all i, and suppose L is a configuration realized by OED forwarding. Suppose I is an even plateau of height h. Then*

$$L_h(I) \leq |I| + \sigma + \sum_{i \in I} \beta(i) \leq (B + 1)\,|I| + \sigma.$$

Proof. For any interval I, let $A_I \subseteq A$ be the injection pattern consisting of packets injected into buffers $i \in I$. Since A is locally (ρ, σ, β)-bounded, A_I is (ρ, σ, β')-bounded with

$$\beta'(i) = \begin{cases} \beta(i) & \text{if } i \in I \\ 0 & \text{otherwise} \end{cases}$$

Therefore, by Observation 2.3, A_I is (ρ, σ')-bounded, where $\sigma' = \sigma + \sum_{i \in I} \beta(i)$.

Now let I be an even plateau of height h. By Corollary 4.6, the configuration L_h (see Equation (7)) consists entirely of packets injected into I – i.e., packets injected by A_I. Since A_I is a (ρ, σ')-bounded adversary, Lemma 4.7 implies that $L_h(I) \leq |I| + \sigma'$, which gives the desired result. ◀

We now have all the pieces together to prove Theorem 4.2. The idea is to use Corollary 4.8 inductively to show that plateaus cannot grow too tall.

Proof of Theorem 4.2. Assume without loss of generality that B is even, and consider any configuration L attained by A. Let $i^* = \arg\max_i L(i)$ be a buffer with maximum load, and define $I_0 \supseteq I_1 \supseteq I_2 \supseteq \cdots \supseteq I_\ell$ where I_j is the plateau of height j containing i^*, and $\ell = L(i^*)$.

Define m to be the maximum value such that

$$\left|I_{m(B+2)}\right| - \frac{\sigma}{B+1} \geq 1, \tag{8}$$

if such a value of $m > 0$ exists, and take $m = 0$ otherwise. Observe that for all k, we have

$$L_k(I_k) \geq \sum_{j=k+1}^{\ell} |I_j|\,. \tag{9}$$

Since the I_j are nested intervals, for any $k < m$ we have

$$L_{k(B+2)}(I_{k(B+2)}) \geq (B+2) \sum_{j=k+1}^{m} L_{j(B+2)}(I_{j(B+2)}). \tag{10}$$

Combining (10) with the result of Corollary 4.8, we find that for all $j < m$,

$$(B+2) \sum_{j=k+1}^{m} \left|I_{j(B+2)}\right| \leq (B+1) \left|I_{k(B+2)}\right| + \sigma. \tag{11}$$

Rearranging (11) yields that for all $k = 0, 1, \ldots m-1$ we have

$$b \sum_{j=k+1}^{m} \left|I_{j(B+2)}\right| - \frac{\sigma}{B+1} \leq \left|I_{k(B+2)}\right| \quad \text{where } b = \frac{B+2}{B+1}. \tag{12}$$

Note that for $k = m-1, m-2, \ldots$, (12) gives

$$b\left|I_{m(B+2)}\right| - \frac{\sigma}{B+1} \leq \left|I_{(m-1)(B+2)}\right| \tag{13}$$

$$b\left|I_{m(B+2)}\right| + b\left|I_{(m-1)(B+2)}\right| - \frac{\sigma}{B+1} \leq \left|I_{(m-2)(B+2)}\right| \tag{14}$$

$$\vdots$$

Combining (13) and (14), we obtain

$$(b+b^2)\left|I_{m(B+2)}\right| - (1+b)\frac{\sigma}{B+1} \leq \left|I_{(m-2)(B+2)}\right|. \tag{15}$$

Continuing in this way, a straightforward induction argument combined with the observation that $b > 1$ gives

$$b^m\left(\left|I_{m(B+2)}\right| - \frac{\sigma}{B+1}\right) \leq |I_0| = n. \tag{16}$$

By the choice of m in (8), (16) implies that

$$m \leq \log_b n. \tag{17}$$

Again, from the definition of m, taking $h = m(B+2)+2$, we have

$$|I_h| < 1 + \frac{\sigma}{B+1}.$$

Applying Corollary 4.8, gives

$$L_h(I_h) \leq (B+1)|I_h| + \sigma \leq B + 2\sigma + 1.$$

Since there are at most $B + 2\sigma + 1$ above I_h, the load of i^* satisfies

$$\begin{aligned} L(i^*) &\leq h + B + 2\sigma + 1 \\ &\leq (B+2)m + B + 2\sigma + 3 \\ &= O(B\log n + \sigma), \end{aligned}$$

which gives the desired result. ◀

4.1 Consequences

Here, we list some consequences of the upper bound of Theorem 4.2 when applied in combination with the packet bundling procedures described in Sections 2.2 and 3. For the following results, we assume that G is an information gathering network with uniform edge capacity C.

▶ **Corollary 4.9.** *Suppose A is a locally (ρ, σ, β)-bounded adversary with $\rho \leq C$ and $\beta(e) \leq B$ for all $e \in E$. Then OED forwarding applied to the C-reduction of A has buffer space usage $O((B + C) \log n + \sigma)$.*

▶ **Corollary 4.10.** *Suppose Φ is a family of flows with locally dependent rate bound $r \leq C$, local burst parameters $b_\phi \leq B$, and global burst parameter σ. Then OED forwarding applied to the discretization of the C-reduction of Φ requires buffer space $O((B + C) \log n + \sigma)$.*

▶ **Corollary 4.11.** *Suppose A is a locally (ρ, σ, β)-bounded adversary with indivisible heterogeneous packet injections, and $\rho \leq C/2$. Then OED forwarding applied to the bundled injection pattern described in Section 3.2 achieves buffer space usage $O((B + C) \log n + \sigma)$.*

5 Lower Bounds on Buffer Size

In this section, we show that buffer space usage of the OED algorithm is asymptotically optimal among deterministic forwarding protocols. In Appendix A, we generalize the lower bound to randomized forwarding protocols.

▶ **Theorem 5.1.** *Let F be any deterministic online forwarding protocol, and let $G = (V, E)$ be a single-destination path of length n. For any B let $\beta(e) = B$ for all $e \in E$. Then there exists a local $(1, \sigma, \beta)$-bounded adversary A such that*

$$\sup_{t,i} L^t(i) = \Omega(B \log_B n + \sigma). \tag{18}$$

▶ Remark 5.2. The lower bound of (18) is in contrast to the centralized and semi-local upper bounds of [8, 11, 9], which show that for (ρ, σ)-bounded adversaries, maximum buffer space $O(\rho + \sigma)$ (with no n dependence) is achievable. In particular, [11] gives a smooth tradeoff between the locality of a forwarding protocol and the optimal buffer space usage. Their work shows that if each node acts based on the state of its d-distance neighborhood, then $\Theta(\frac{1}{d} \log n + \sigma)$ buffer space is necessary and sufficient.[3] Thus, for (ρ, σ)-bounded adversaries, the worst-case buffer space usage for a protocol generally depends on the protocol's locality (d). Together, Theorems 4.2 and 5.1 show that this is not the case for local (ρ, σ, β)-bounded adversaries, as OED – a local ($d = O(1)$) protocol – is asymptotically optimal, even compared to centralized protocols. Thus, unlike for (ρ, σ)-bounded injection patterns, non-local information does not asymptotically improve the performance against local (ρ, σ, β)-bounded adversaries in information gathering networks.

▶ Remark 5.3. In the proof of Theorem 5.1, we describe an injection pattern as defined by an *adaptive offline adversary*. That is, the choices made by the adversary are made in response to an algorithm's forwarding decisions (i.e., the current state of all buffers in the network). If the forwarding protocol is deterministic, this assumption about the adversary is without loss of generality, as the adversary can simulate the forwarding protocol and

[3] In the case $d = 1$, the algorithm of [11] reduces to the OED algorithm.

construct an injection pattern in advance. In Appendix A, we will describe a (randomized) *oblivious* adversary that is unaware of the forwarding protocol being used. Nonetheless, the oblivious adversary will almost surely require buffer space usage of $\Omega(B \log n + \sigma)$ against any (centralized, randomized) online forward protocol.

Proof of Theorem 5.1. Without loss of generality, we assume that the network size is a power of $2B$, say, $n = (2B)^m$. Given a deterministic, online forwarding protocol F, we construct an adversary A_F that injects packets in phases. Specifically, A_F chooses a nested sequence of sub-intervals $I_0 \supseteq I_1 \supseteq I_2 \supseteq \cdots$ and in the kth phase, A_F injects packets only into I_k. Each I_k is chosen at the end of the $(k-1)^{\text{st}}$ phase depending on the loads of buffers in I_{k-1}. The kth phase lasts τ_k rounds.

We set $\tau_1 = \frac{1}{2}n$ and $I_1 = [n]$. Inductively, we define $\tau_k = \frac{1}{2B}\tau_{k-1}$ and $|I_k| = \frac{1}{2B}|I_{k-1}|$. At the beginning of phase k, A_F selects I_k and injects B packets into each buffer in I_k. For $k > 1$, A_F selects I_k as follows. Let

$$I_{k-1} = I_{k-1}^1 \cup I_{k-1}^2 \cup \cdots \cup I_{k-1}^{2B},$$

where the I_{k-1}^j are consecutive intervals of size $\frac{1}{2B}|I_{k-1}|$. Then A_F selects $I_k = I_{k-1}^j$ where I_{k-1}^j has the largest total load at the end of the kth phase. That is, $j = \arg\max_j L(I_{k-1}^j)$. Observe that for all k we have

$$\tau_k = \frac{1}{2}|I_k|, \text{ and} \tag{19}$$

$$\tau_k = B|I_{k+1}|. \tag{20}$$

▷ Claim. For all k, at the end of the kth phase, the total load of I_k satisfies

$$L(I_k) \geq k\left(B - \frac{1}{2}\right)|I_k|. \tag{21}$$

Proof of Claim. We argue by induction on k. For $k = 1$, at the beginning of the first phase, B packets are injected into each buffer in the network. Thus, the total load is $Bn = B|I_1|$. After $|\tau_1| = \frac{1}{2}n$ forwarding rounds, at most $\frac{1}{2}n = \frac{1}{2}|I_1|$ packets are forwarded by the last buffer in I_1, hence the total load in I_1 is at least $(B - 1/2)|I_1|$.

For the inductive step, assume that $L(I_{k-1}) \geq (k-1)(B-1/2)|I_{k-1}|$ at the end of the $(k-1)^{\text{st}}$ phase. By the choice of I_k, we therefore have $L(I_k) \geq (k-1)(B-1/2)|I_k|$ at the end of the $(k-1)^{\text{st}}$ phase. At the beginning of the kth phase, A_F injects $B|I_k|$ packets into the buffers in I_k, hence the load becomes at least $(kB - (k-1)/2)|I_k|$. During the $\tau_k = |I_k|/2$ rounds of the kth phase, the last buffer in I_k forwards at most $\tau_k = |I_k|/2$ packets, hence the total load of I_k decreases by at most this amount. Thus we have $k(B-1/2)|I_k|$ as desired. ◁

Applying the claim, after $k = \log_{2B} n$ phases, we have $|I_k| = 1$ and $L(I_k) = k(B-1/2) = \Omega(B \log_B n))$. The desired result follows by adding one more round in which A_F injects σ packets into I_k.

All that remains is to show that A_F is local $(1, \sigma, \beta)$-bounded. To this end, suppose each kth phase begins in round s_k and ends in round t_k. Then injections only occur in rounds s_k. Now fix any subset S of nodes and interval $T = [s, t]$, and define j and ℓ such that $s_{j-1} < s \leq t_j$, and ℓ with $s_\ell \leq t \leq t_\ell$. For k satisfying $j \leq k \leq \ell$, define $S_k = S \cap I_k$. Then observe that in round s_k, $B|S_k|$ packets are injected into S. Therefore, the total number of packets injected into S during T is $N = \sum_{k=j}^{\ell} B|S_k| \leq \sum_{k=j}^{\ell} B|I_k|$, while the total number of rounds is $|T| \geq \max\left\{1, \sum_{k=j+1}^{\ell-1} \tau_k\right\}$. We bound N as follows:

$$
\begin{aligned}
N &= B\,|S_j| + B\,|S_{j+2}| + \cdots + B\,|S_\ell| \\
&\le B\,|S_j| + B\,|I_{j+2}| + \cdots + B\,|I_\ell| \\
&= B\,|S_j| + \tau_{j+1} + \cdots + \tau_{\ell-1} \\
&\le B\,|S| + |T|\,.
\end{aligned}
$$

The second equality comes from Equation (20). Thus, A_F is local $(1, \sigma, \beta)$ bounded. ◀

In Appendix A, we generalize the lower bound to randomized protocols.

References

1 Micah Adler and Adi Rosén. Tight bounds for the performance of longest-in-system on dags. In Helmut Alt and Afonso Ferreira, editors, *STACS 2002: 19th Annual Symposium on Theoretical Aspects of Computer Science, Antibes - Juan les Pins, France, March 14–16, 2002 Proceedings*, pages 88–99, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. doi:10.1007/3-540-45841-7_6.

2 Allan Borodin, Jon Kleinberg, Prabhakar Raghavan, Madhu Sudan, and David P. Williamson. Adversarial queuing theory. *J. ACM*, 48(1):13–38, January 2001. doi:10.1145/363647.363659.

3 Johannes Bund, Matthias Függer, Christoph Lenzen, Moti Medina, and Will Rosenbaum. PALS: plesiochronous and locally synchronous systems. In *26th IEEE International Symposium on Asynchronous Circuits and Systems, ASYNC 2020, Salt Lake City, UT, USA, May 17-20, 2020*, pages 36–43. IEEE, 2020. doi:10.1109/ASYNC49171.2020.00013.

4 R. L. Cruz. A calculus for network delay, part I: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, January 1991. doi:10.1109/18.61109.

5 R.L. Cruz. A calculus for network delay. ii. network analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, 1991. doi:10.1109/18.61110.

6 Stefan Dobrev, Manuel Lafond, Lata Narayanan, and Jaroslav Opatrny. Optimal local buffer management for information gathering with adversarial traffic. In Christian Scheideler and Mohammad Taghi Hajiaghayi, editors, *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2017, Washington DC, USA, July 24-26, 2017*, pages 265–274. ACM, 2017. doi:10.1145/3087556.3087577.

7 S. Kundu and S. Chattopadhyay. *Network-on-Chip: The Next Generation of System-on-Chip Integration*. CRC Press, 2018.

8 Avery Miller and Boaz Patt-Shamir. Buffer size for routing limited-rate adversarial traffic. In *DISC 2016: Proceedings of the 30th International Symposium on Distributed Computing, Paris, France, September 27-29, 2016*, pages 328–341. Springer, 2016. doi:10.1007/978-3-662-53426-7_24.

9 Avery Miller, Boaz Patt-Shamir, and Will Rosenbaum. With great speed come small buffers: Space-bandwidth tradeoffs for routing. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 117–126. ACM, 2019. doi:10.1145/3293611.3331614.

10 Boaz Patt-Shamir and Will Rosenbaum. The space requirement of local forwarding on acyclic networks. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *PODC 2017: Proceedings of the ACM Symposium on Principles of Distributed Computing, Washington, DC, USA, July 25-27, 2017*, pages 13–22. ACM, 2017. doi:10.1145/3087801.3087803.

11 Boaz Patt-Shamir and Will Rosenbaum. Space-optimal packet routing on trees. In *2019 IEEE Conference on Computer Communications, INFOCOM 2019, Paris, France, April 29 - May 2, 2019*, pages 1036–1044. IEEE, 2019. doi:10.1109/INFOCOM.2019.8737596.

12 David Salyers, Yingxin Jiang, Aaron Striegel, and Christian Poellabauer. Jumbogen: Dynamic jumbo frame generation for network performance scalability. *SIGCOMM Comput. Commun. Rev.*, 37(5):53–64, October 2007. doi:10.1145/1290168.1290174.

13 Stefan Schmid and Jukka Suomela. Exploiting locality in distributed sdn control. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 121–126, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2491185.2491198.

A Generalization to Randomized Protocols

Here, we generalize the lower bound of Theorem 5.1 to *randomized* forwarding protocols. Specifically, we construct a randomized *oblivious* injection pattern that requires buffer space $\Omega(B \log n + \sigma)$ against any online forwarding protocol.

▶ **Theorem A.1.** *Let $G = (V, E)$ be the single destination path of length n. Then for every σ, and B, there exists a randomized injection pattern $A_{\text{rand}} \in \mathcal{L}(1, \sigma, B)$ such that for any (centralized, randomized) online forwarding protocol F, we have*

$$\sup_{t,i} L^t(i) = \Omega(B \log_B n + \sigma)$$

almost surely.

Again, we emphasize the order of quantifiers in the statement of the Theorem A.1: the *same* randomized injection pattern achieves the lower bound (almost surely) for every forwarding protocol.

The adversary A_{rand} of Theorem A.1 is a straightforward modification of the adversary A_F constructed the proof of Theorem 5.1. Recall that A_F injects packets into a nested sequence of intervals $I_0 \supseteq I_1 \supseteq \cdots \supseteq I_k$ where $k = O(\log_B n)$, and $|I_k| = 1$. Each for $j \geq 1$, I_j is chosen to be one of $2B$ sub-intervals of I_{j-1} with maximum average load and $|I_{j-1}| = 2B\,|I_j|$. The idea of A_{rand} is to perform the same injection pattern as A, except that A_{rand} chooses each $I_j \subseteq I_{j-1}$ randomly, independent of the choices of the forwarding protocol. We will show that for any execution of any forwarding protocol, injecting in this way yields a load of $\Omega(B \log n + \sigma)$ with probability $\Omega(1/n)$. Thus, by independently repeating the randomized injection pattern ad infinitum, the lower bound is achieved almost surely (and with high probability after $O(n^2)$ injection rounds).

In order to formalize our description of A_{rand}, we first observe that the sequence $I_0 \supseteq I_1 \supseteq \cdots \supseteq I_k$ of intervals chosen by A_F is uniquely determined by $I_k = [a_k]$, the final buffer into which A_F injects packets. We also note that the injection pattern A_F consists of $O(n)$ injection rounds, in which $O(Bn + \sigma)$ packets in total are injected. Let A_i denote such an injection pattern in which $I_k = [i]$ – i.e., i is the final buffer into which A_i injects packets.

▶ **Definition A.2.** *Let A_i be the injection pattern described in the preceding paragraph. Then the adversary A_{rand} injects packets as follows. Repeat:*

1. *choose $i \in [n]$ uniformly at random.*
2. *in $O(n)$ rounds, inject packets as in A_i*
3. *wait $Bn + \sigma$ rounds without injecting any packets*

A single iteration of step 1–3 is an ***epoch****, and each epoch consists of $k = O(\log n)$* ***phases*** *(corresponding to the sub-intervals I_j chosen in A_i).*

▶ **Definition A.3.** *Consider an execution of some forwarding protocol F with adversary A_{rand}. We say that phase $j > 0$ of some epoch of A_{rand} is* ***good*** *if at the beginning of the phase we have*

$$\frac{L(I_j)}{|I_j|} \geq \frac{L(I_{j-1})}{|I_{j-1}|}.$$

We say that an epoch is ***good*** *if all of its phases are good.*

The following corollary follows immediately from the proof of Theorem 5.1.

▶ **Corollary A.4.** *Suppose an execution of a protocol F with adversary A_{rand} experiences a good epoch with injection pattern A_i. Then in the epoch's final injection round, $L(i) = \Omega(B \log n + \sigma)$.*

By Corollary A.4, all that remains to prove Theorem A.1 is to show that each epoch is good with sufficient probability.

▶ **Lemma A.5.** *Consider a single epoch of an execution of a protocol F with adversary A_{rand}. Then each phase is good independently with probability at least $1/2B$.*

Proof. Consider the interval I_{j-1} at the beginning of the jth phase. There are $2B$ choices of the sub-interval I_j, and each is chosen with equal probability. By the pigeonhole principle, at least one choice I_j is good. Since A_i is chosen uniformly at random with $i \in [n]$, conditioned on I_{j-1}, the choice of I_j is uniformly at random. Thus, the probability I_j is a good choice is at least $1/2B$. Finally, we note that since i is chosen uniformly at random, the choice of *which sub-interval* of I_{j-1} is chosen in phase j is independent of the sub-intervals chosen in other phases. ◀

▶ **Corollary A.6.** *Each epoch is good independently with probability at least $1/n$.*

Proof. Each epoch consists of $\log_{2B} n$ phases, and each phase is good independently with probability $1/2B$. Therefore, the probability that all phases are good is $(1/2B)^{\log_{2B} n} = 1/n$. By construction, the choices of i used for each epoch are mutually independent. ◀

Proof of Theorem A.1. The proof of Theorem 5.1 implies that each A_i chosen by A_{rand} is locally (ρ, σ, β)-bounded. Waiting $Bn + \sigma$ rounds with no injections between epochs ensures that A_{rand} is locally (ρ, σ, β)-bounded as well.

By Corollary A.4, if A_{rand} experiences a good epoch, then it inflicts a buffer load of $\Omega(B \log n+\sigma)$. By Corollary A.6, the probability that the first k epochs are not good is at most $(1-1/n)^k$. Taking the limit as $k \to \infty$, we find that $\Pr(\sup_{t,i'} L^t(i') = o(B \log n+\sigma)) = 0$. ◀

▶ Remark A.7. The proof of Theorem A.1 shows that the probability that none of the first k epochs are good is at most $(1 - 1/n)^k$. This expression implies that a good epoch occurs with high probability after $k = O(n \log n)$ epochs. Since each epoch lasts $O(Bn + \sigma)$ rounds, a good epoch (hence a load of $\Omega(B \log n + \sigma)$) occurs after $O(Bn^2 \log n)$ rounds with high probability.

The Weakest Failure Detector for Genuine Atomic Multicast

Pierre Sutra
Telecom SudParis, Palaiseau, France

Abstract

Atomic broadcast is a group communication primitive to order messages across a set of distributed processes. Atomic multicast is its natural generalization where each message m is addressed to $dst(m)$, a subset of the processes called its destination group. A solution to atomic multicast is *genuine* when a process takes steps only if a message is addressed to it. Genuine solutions are the ones used in practice because they have better performance.

Let $\mathcal{G}$ be all the destination groups and $\mathcal{F}$ be the cyclic families in it, that is the subsets of $\mathcal{G}$ whose intersection graph is hamiltonian. This paper establishes that the weakest failure detector to solve genuine atomic multicast is $\mu = (\wedge_{g,h \in \mathcal{G}}\ \Sigma_{g \cap h}) \wedge (\wedge_{g \in \mathcal{G}}\ \Omega_g) \wedge \gamma$, where Σ_P and Ω_P are the quorum and leader failure detectors restricted to the processes in P, and γ is a new failure detector that informs the processes in a cyclic family $\mathfrak{f} \in \mathcal{F}$ when $\mathfrak{f}$ is faulty.

We also study two classical variations of atomic multicast. The first variation requires that message delivery follows the real-time order. In this case, μ must be strengthened with $1^{g \cap h}$, the indicator failure detector that informs each process in $g \cup h$ when $g \cap h$ is faulty. The second variation requires a message to be delivered when the destination group runs in isolation. We prove that its weakest failure detector is at least $\mu \wedge (\wedge_{g,h \in \mathcal{G}}\ \Omega_{g \cap h})$. This value is attained when $\mathcal{F} = \varnothing$.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Software and its engineering → Distributed systems organizing principles; General and reference → Reliability

Keywords and phrases Failure Detector, State Machine Replication, Consensus

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.35

Related Version *Extended Version*: https://arxiv.org/abs/2208.07650 [37]

1 Introduction

Context. Multicast is a fundamental group communication primitive used in modern computing infrastructures. This primitive allows to disseminate a message to a subset of the processes in the system, its destination group. Implementations exist over point-to-point protocols such as the Internet Protocol. Multicast is atomic when it offers the properties of atomic broadcast to the multicast primitive: each message is delivered at most once, and delivery occurs following some global order. Atomic multicast is used to implement strongly consistent data storage [4, 11, 36, 32].

It is easy to see that atomic multicast can be implemented atop atomic broadcast. Each message is sent through atomic broadcast and delivered where appropriate. Such a naive approach is however used rarely in practice because it is inefficient when the number of destination groups is large [31, 35]. To rule out naive implementations, Guerraoui and Schiper [25] introduce the notion of genuineness. An implementation of atomic multicast is *genuine* when a process takes steps only if a message is addressed to it.

Existing genuine atomic multicast algorithms that are fault-tolerant have strong synchrony assumptions on the underlying system. Some protocols (such as [34]) assume that a perfect failure detector is available. Alternatively, a common assumption is that the destination groups are decomposable into disjoint groups, each of these behaving as a logically correct entity. Such an assumption is a consequence of the impossibility result established in [25].

© Pierre Sutra;
licensed under Creative Commons License CC-BY 4.0

Table 1 About the weakest failure detector for atomic multicast. *(√√= strongly genuine)*

Genuiness	Order	Weakest	
×	Global	$\Omega \wedge \Sigma$	[8, 15]
√	·	$\notin \mathcal{U}_2$	[25]
·	·	$\leq P$	[34]
·	·	μ	§5, §4
·	Strict	$\mu \wedge (\wedge_{g,h\in\mathcal{G}}\ 1^{g\cap h})$	§6.1
·	Pairwise	$(\wedge_{g,h\in\mathcal{G}}\ \Sigma_{g\cap h}) \wedge (\wedge_{g\in\mathcal{G}}\ \Omega_g)$	§7
√√	Global	**if** $\mathcal{F} = \varnothing$ **then** $\mu \wedge (\wedge_{g,h\in\mathcal{G}}\ \Omega_{g\cap h})$ **else** $\geq \mu \wedge (\wedge_{g,h\in\mathcal{G}}\ \Omega_{g\cap h})$	§6.2

This result states that genuine atomic multicast requires some form of perfect failure detection in intersecting groups. Consequently, almost all protocols published to date (e.g., [30, 17, 20, 10, 29, 13]) assume the existence of such a decomposition.

Motivation. A key observation is that the impossibility result in [25] is established when atomic multicast allows a message to be disseminated to *any* subset of the processes. However, where there is no such need, weaker synchrony assumptions may just work. For instance, when each message is addressed to a single process, the problem is trivial and can be solved in an asynchronous system. Conversely, when every message is addressed to all the processes in the system, atomic multicast boils down to atomic broadcast, and thus ultimately to consensus. Now, if no two groups intersect, solving consensus inside each group seems both necessary and sufficient. In this paper, we further push this line of thought to characterize the necessary and sufficient synchrony assumptions to solve genuine atomic multicast.

Our results are established in the unreliable failure detectors model [9, 18]. A failure detector is an oracle available locally to each process that provides information regarding the speed at which the other processes are taking steps. Finding the weakest failure detector to solve a given problem is a central question in distributed computing literature [18]. In particular, the seminal work in [8] shows that a leader oracle (Ω) is the weakest failure detector for consensus when a majority of processes is correct. If any processes might fail, then a quorum failure detector (Σ) is required in addition to Ω [15].

A failure detector is realistic when it cannot guess the future. In [14], the authors prove that the perfect failure detector (P) is the weakest realistic failure detector to solve consensus. Building upon this result, Schiper and Pedone [34] shows that P is sufficient to implement genuine atomic multicast. However, P is the weakest only when messages are addressed to all the processes in the system. The present paper generalizes this result and the characterization given in [25] (see Table 1). It establishes the weakest failure detector to solve genuine atomic multicast for any set of destination groups.

Primer on the findings. Let $\mathcal{G}$ be all the destination groups and $\mathcal{F}$ be the cyclic families in it, that is the subsets of $\mathcal{G}$ whose intersection graph is hamiltonian. This paper shows that the weakest failure detector to solve genuine atomic multicast is $\mu = (\wedge_{g,h\in\mathcal{G}}\ \Sigma_{g\cap h}) \wedge (\wedge_{g\in\mathcal{G}}\ \Omega_g) \wedge \gamma$, where Σ_P and Ω_P are the quorum and leader failure detectors restricted to the processes in P, and γ is a new failure detector that informs the processes in a cyclic family $\mathfrak{f} \in \mathcal{F}$ when $\mathfrak{f}$ is faulty. Our results regarding γ are established wrt. realistic failure detectors.

This paper also studies two classical variations of the atomic multicast problem. The strict variation requires that message delivery follows the real-time order. In this case, we prove that μ must be strengthen with $1^{g\cap h}$, the indicator failure detector that informs each

process in $g \cup h$ when $g \cap h$ is faulty. The strongly genuine variation requires a message to be delivered when its destination group runs in isolation. In that case, the weakest failure detector is at least $\mu \wedge (\wedge_{g,h \in \mathcal{G}} \Omega_{g \cap h})$. This value is attained when $\mathcal{F} = \varnothing$.

Outline of the paper. §2 introduces the atomic multicast problem and the notion of genuineness. We present the candidate failure detector in §3. §4 proves that this candidate is sufficient. Its necessity is established in §5. §6 details the results regarding the two variations of the problem. We cover related work and discuss our results in §7. §8 closes this paper. Due to space constraints, all the proofs are deferred to the extended version [37].

2 The Atomic Multicast Problem

2.1 System Model

In [9], the authors extend the usual model of asynchronous distributed computation to include failure detectors. The present paper follows this model with the simplifications introduced in [23, 22]. This model is recalled in [37].

2.2 Problem Definition

Atomic multicast is a group communication primitive that allows to disseminate messages between processes. This primitive is used to build transactional systems [11, 36] and partially-replicated (aka., sharded) data stores [17, 32]. In what follows, we consider the most standard definition for this problem [4, 26, 12]. In the parlance of Hadzilacos and Toueg [26], it is named uniform global total order multicast. Other variations are studied in §6.

Given a set of messages $\mathcal{M}$, the interface of atomic multicast consists of operations *multicast*(m) and *deliver*(m), with $m \in \mathcal{M}$. Operation *multicast*(m) allows a process to *multicast* a message m to a set of processes denoted by *dst*(m). This set is named the *destination group* of m. When a process executes *deliver*(m), it delivers message m, typically to an upper applicative layer.

Consider two messages m and m' and some process $p \in dst(m) \cap dst(m')$. Relation $m \overset{p}{\mapsto} m'$ captures the local delivery order at process p. This relation holds when, at the time p delivers m, p has not delivered m'. The union of the local delivery orders gives the *delivery order*, that is $\mapsto = \cup_{p \in \mathcal{P}} \overset{p}{\mapsto}$. The runs of atomic multicast must satisfy:

(Integrity) For every process p and message m, p delivers m at most once, and only if p belongs to *dst*(m) and m was previously multicast.

(Termination) For every message m, if a correct process multicasts m, or a process delivers m, eventually every correct process in *dst*(m) delivers m.

(Ordering) The transitive closure of $\mapsto$ is a strict partial order over $\mathcal{M}$.

Integrity and termination are two common properties in group communication literature. They respectively ensure that only sound messages are delivered to the upper layer and that the communication primitive makes progress. Ordering guarantees that the messages could have been received by a sequential process. A common and equivalent rewriting of this property is as follows:

(Ordering) Relation $\mapsto$ is acyclic over $\mathcal{M}$.

If the sole destination group is $\mathcal{P}$, that is the set of all the processes, the definition above is the one of atomic broadcast. In what follows, $\mathcal{G} \subseteq 2^{\mathcal{P}}$ is the set of all the destinations groups, i.e., $\mathcal{G} = \{g : \exists m \in \mathcal{M}. \, g = dst(m)\}$. For some process p, $\mathcal{G}(p)$ denotes the destination groups in $\mathcal{G}$ that contain p. Two groups g and h are *intersecting* when $g \cap h \neq \varnothing$.

What can be sent and to who. The process that executes $multicast(m)$ is the sender of m, denoted $src(m)$. As usual, we consider that processes disseminate different messages (i.e., src is a function). A message holds a bounded payload $payload(m)$, and we assume that atomic multicast is not payload-sensitive. This means that for every message m, and for every possible payload x, there exists a message $m' \in \mathcal{M}$ such that $payload(m') = x$, $dst(m') = dst(m)$ and $src(m') = src(m)$.

Dissemination model. In this paper, we consider a closed model of dissemination. This means that to send a message to some group g, a process must belong to it (i.e., $src(m) \in dst(m)$). In addition, we do not restrict the source of a message. This translates into the fact that for every message m, for every process p in $dst(m)$, there exists a message m' with $dst(m) = dst(m')$ and $src(m') = p$. Under the above set of assumptions, the atomic multicast problem is fully determined by the destination groups $\mathcal{G}$.

2.3 Genuineness

At first glance, atomic multicast boils down to the atomic broadcast problem: to disseminate a message it suffices to broadcast it, and upon reception only messages addressed to the local machine are delivered. With this approach, every process takes computational steps to deliver every message, including the ones it is not concerned with. As a consequence, the protocol does not scale [31, 35], even if the workload is embarrassingly parallel (e.g., when the destinations groups are pairwise disjoint).

Such a strategy defeats the core purpose of atomic multicast and is thus not satisfying from a performance perspective. To rule out this class of solutions, Guerraoui and Schiper [25] introduce the notion of *genuine* atomic multicast. These protocols satisfy the minimality property defined below.

(Minimality) In every run R of A, if some correct process p sends or receives a (non-null) message in R, there exists a message m multicast in R with $p \in dst(m)$.

All the results stated in this paper concern genuine atomic multicast. To date, this is the most studied variation for this problem (see, e.g., [30, 20, 10]).

3 The Candidate Failure Detector

This paper characterizes the weakest failure detector to solve genuine atomic multicast. Below, we introduce several notions related to failure detectors then present our candidate.

Family of destination groups. A family of destination groups is a set of (non-repeated) destination groups $\mathfrak{f} = (g_i)_i$. For some family $\mathfrak{f}$, $cpaths(\mathfrak{f})$ are the closed paths in the intersection graph of $\mathfrak{f}$ visiting all its destination groups.[1] Family $\mathfrak{f}$ is *cyclic* when its intersection graph is hamiltonian, that is when $cpaths(\mathfrak{f})$ is non-empty. A cyclic family $\mathfrak{f}$ is *faulty at time* t when every path $\pi \in cpaths(\mathfrak{f})$ visits an edge (g, h) with $g \cap h$ faulty at t.

[1] The intersection graph of a family of sets $(S_i)_i$ is the undirected graph whose vertices are the sets S_i, and such that there is an edge linking S_i and S_j iff $S_i \cap S_j \neq \varnothing$.

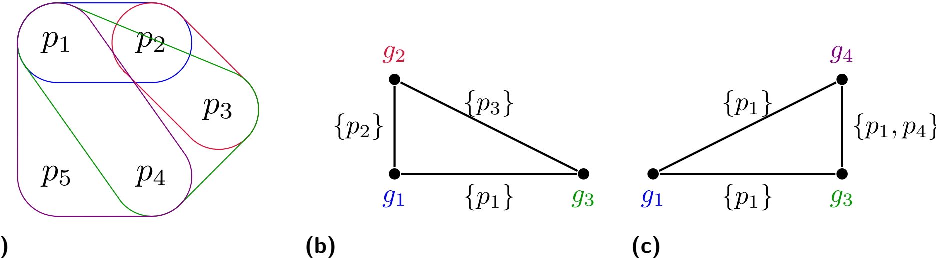


Figure 1 From left to right: the four groups g_1, g_2, g_3 and g_4, and the intersection graphs of the two cyclic families $\mathfrak{f} = \{g_1, g_2, g_3\}$ and $\mathfrak{f}' = \{g_1, g_3, g_4\}$.

In what follows, $\mathcal{F}$ denotes all the cyclic families in $2^{\mathcal{G}}$. Given a destination group g, $\mathcal{F}(g)$ are the cyclic families in $\mathcal{F}$ that contain g. For some process p, $\mathcal{F}(p)$ are the cyclic families $\mathfrak{f}$ such that p belongs to some group intersection in $\mathfrak{f}$ (that is, $\exists g, h \in \mathfrak{f}.\, p \in g \cap h$).

To illustrate the above notions, consider Figure 1. In this figure, $\mathcal{P} = \{p_1, \ldots, p_5\}$ and we have four destination groups: $g_1 = \{p_1, p_2\}$, $g_2 = \{p_2, p_3\}$, $g_3 = \{p_1, p_3, p_4\}$ and $g_4 = \{p_1, p_4, p_5\}$. The intersection graphs of $\mathfrak{f} = \{g_1, g_2, g_3\}$ and $\mathfrak{f}' = \{g_1, g_3, g_4\}$ are depicted respectively in Figures 1b and 1c. These two families are cyclic. This is also the case of $\mathfrak{f}'' = \mathcal{G} = \{g_1, g_2, g_3, g_4\}$ whose intersection graph is the union of the two intersection graphs of $\mathfrak{f}$ and $\mathfrak{f}'$. This family is faulty when $g_2 \cap g_1 = \{p_2\}$ fails. Group g_2 belongs to two cyclic families, namely $\mathcal{F}(g_2) = \{\mathfrak{f}, \mathfrak{f}''\}$. Process p_1 belongs to all cyclic families, that is $\mathcal{F}(p_1) = \mathcal{F}$. Differently, since p_5 is part of no group intersection, $\mathcal{F}(p_5) = \varnothing$.

Failure detectors of interest. Failure detectors are grouped into classes of equivalence that share common computational power. Several classes of failure detectors have been proposed in the past. This paper makes use of two common classes of failure detectors, Σ and Ω, respectively introduced in [15] and [8]. We also propose a new class γ named the *cyclicity failure detector*. All these classes are detailed below.

- The quorum failure detector (Σ) captures the minimal amount of synchrony to implement an atomic register. When a process p queries at time t a detector of this class, it returns a non-empty subset of processes $\Sigma(p,t) \subseteq \mathcal{P}$ such that:
  (Intersection) $\forall p, q \in \mathcal{P}.\, \forall t, t' \in \mathbb{N}.\, \Sigma(p,t) \cap \Sigma(q,t') \neq \varnothing$
  (Liveness) $\forall p \in Correct.\, \exists \tau \in \mathbb{N}.\, \forall t \geq \tau.\, \Sigma(p,t) \subseteq Correct$
  The first property states that the values of any two quorums taken at any times intersect. It is used to maintain the consistency of the atomic register. The second property ensures that eventually only correct processes are returned.
- Failure detector Ω returns an eventually reliable leader [16]. In detail, it returns a value $\Omega(p,t) \in \mathcal{P}$ satisfying that:
  (Leadership) $Correct \neq \varnothing \Rightarrow (\exists l \in Correct.\, \forall p \in Correct.\, \exists \tau \in \mathbb{N}.\, \forall t \geq \tau.\, \Omega(p,t) = l)$
  Ω is the weakest failure detector to solve consensus when processes have access to a shared memory. For message-passing distributed systems, $\Omega \wedge \Sigma$ is the weakest failure detector.
- The cyclicity failure detector (γ) informs each process of the cyclic families it is currently involved with. In detail, failure detector γ returns at each process p a set of cyclic families $\mathfrak{f} \in \mathcal{F}(p)$ such that:
  (Accuracy) $\forall p \in \mathcal{P}.\, \forall t \in \mathbb{N}.\, (\mathfrak{f} \in \mathcal{F}(p) \wedge \mathfrak{f} \notin \gamma(p,t)) \Rightarrow \mathfrak{f}$ faulty at t
  (Completeness) $\forall p \in Correct.\, \forall t \in \mathbb{N}.\, (\mathfrak{f} \in \mathcal{F}(p) \wedge \mathfrak{f}$ faulty at $t) \Rightarrow \exists \tau \in \mathbb{N}.\, \forall t' \geq \tau.\, \mathfrak{f} \notin \gamma(p,t')$

Accuracy ensures that if some cyclic family $\mathfrak{f}$ is not output at p and p belongs to it, then $\mathfrak{f}$ is faulty at that time. Completeness requires that eventually γ does not output forever a faulty family at the correct processes that are part of it. Hereafter, $\gamma(g)$ denotes the groups h such that $g \cap h \neq \varnothing$ and g and h belong to a cyclic family output by γ.

To illustrate the above definitions, we may consider again the system depicted in Figure 1. Let us assume that $Correct = \{p_1, p_4, p_5\}$. The quorum failure detector Σ can return g_1 or g_3, then g_4 forever. Failure detector Ω may output any process, then at some point in time, one of the correct processes (e.g., p_1) ought to be elected forever. At processes p_1, γ returns initially $\{\mathfrak{f}, \mathfrak{f}', \mathfrak{f}''\}$. Then, once families $\mathfrak{f}$ and $\mathfrak{f}''$ are faulty – this should happen as p_2 is faulty – the output eventually stabilizes to $\{\mathfrak{f}'\}$. When this happens, $\gamma(g_1) = \{g_3, g_4\}$.

Conjunction of failure detectors. We write $C \wedge D$ the conjunction of the failure detectors C and D [23]. For a failure pattern F, failure detector $C \wedge D$ returns a history in $D(F) \times C(F)$.

Set-restricted failure detectors. For some failure detector D, D_P is the failure detector obtained by restricting D to the processes in $P \subseteq \mathcal{P}$. This failure detector behaves as D for the processes $p \in P$, and it returns $\perp$ at $p \notin P$. In detail, let $F \cap P$ be the failure pattern F obtained from F by removing the processes outside P, i.e., $(F \cap P)(t) = F(t) \cap P$. Then, $D_P(F)$ equals $D(F \cap P)$ at $p \in P$, and the mapping $p \times \mathbb{N} \rightarrow \perp$ elsewhere. To illustrate this definition, $\Omega_{\{p\}}$ is the trivial failure detector that returns p at process p. Another example is given by $\Sigma_{\{p_1,p_2\}}$ which behaves as Σ over $\mathcal{P} = \{p_1, p_2\}$.

The candidate. Our candidate failure detector is $\mu_{\mathcal{G}} = (\wedge_{g,h \in \mathcal{G}} \ \Sigma_{g \cap h}) \wedge (\wedge_{g \in \mathcal{G}} \ \Omega_g) \wedge \gamma$. When the set of destinations groups $\mathcal{G}$ is clear from the context, we shall omit the subscript.

4 Sufficiency

This section shows that genuine atomic multicast is solvable with the candidate failure detector. A first observation toward this result is that consensus is wait-free solvable in g using $\Sigma_g \wedge \Omega_g$. Indeed, Σ_g permits to build shared atomic registers in g [15]. From these registers, we may construct an obstruction-free consensus and boost it with Ω_g [24]. Thus, any linearizable wait-free shared objects is implementable in g [27]. Leveraging these observations, this section depicts a solution built atop (high-level) shared objects.

Below, we first introduce group sequential atomic multicast (§4.1). From a computability perspective, this simpler variation is equivalent to the common atomic multicast problem. This is the variation that we shall implement hereafter. We then explain at coarse grain how to solve genuine atomic multicast in a fault-tolerant manner using the failure detector μ (§4.2). Further, the details of our solution are presented and its correctness informally argued (§4.3).

4.1 A simpler variation

Group sequential atomic multicast requires that each group handles its messages sequentially. In detail, given two messages m and m' addressed to the same group, we write $m \prec m'$ when $src(m')$ delivers m before it multicasts m'. This variation requires that if m and m' are multicast to the same group, then $m \prec m'$, or the converse, holds. Proposition 1 below establishes that this variation is as difficult as (vanilla) atomic multicast. Building upon this insight, this section depicts a solution to group sequential atomic multicast using failure detector μ.

▶ **Proposition 1.** *Group sequential atomic multicast is equivalent to atomic multicast.*

4.2 Overview of the solution

First of all, we observe that if the groups are pairwise disjoint, it suffices that each group orders the messages it received to solve atomic multicast. To this end, we use a shared log LOG_g per group g. Then, consider that the intersection graph of $\mathcal{G}$ is acyclic, i.e., $\mathcal{F}$ is empty, yet groups may intersect. In that case, it suffices to add a deterministic merge procedure in each group intersection, for instance, using a set of logs $\text{LOG}_{g \cap h}$ when $g \cap h \neq \varnothing$.

Now, to solve the general case, cycles in the order built with the shared logs must be taken into account. To this end, we use a fault-tolerant variation of Skeen's solution [5, 21]: in each log, the message is bumped to the highest initial position it occupies in all the logs. In the original algorithm [5], as in many other approaches (e.g., [20, 10]), such a procedure is failure-free, and processes simply agree on the final position (aka., timestamp) of the message in the logs. In contrast, our algorithm allows a disagreement when the cyclic family becomes faulty. This disagreement is however restricted to different logs, as in the acyclic case.

4.3 Algorithm

Algorithm 1 depicts a solution to (group sequential) genuine atomic multicast using failure detector μ. To the best of our knowledge, this is only algorithm with [34] that tolerates arbitrary failures. Algorithm 1 is composed of a set of actions. An action is executable once its preconditions (**pre:**) are true. The effects (**eff:**) of an action are applied sequentially until it returns. Algorithm 1 uses a log per group and per group intersection. Logs are linearizable, long-lived and wait-free. Their sequential interface is detailed below.

Logs. A *log* is an infinite array of slots. Slots are numbered from 1. Each slot contains one or more data items. A datum d is at position k when slot k contains it. This position is obtained through a call to $pos(d)$; 0 is returned if d is absent. A slot k is *free* when it contains no data item. In the initial state, every slot is free. The head of the log points to the first free slot after which there are only free slots (initially, slot 1). Operation $append(d)$ inserts datum d at the slot pointed by the head of the log then returns its position. If d is already in the log, this operation does nothing. When d is in the log, it can be locked with operation $bumpAndLock(d, k)$. This operation moves d from its current slot l to slot $\max(k, l)$, then locks it. Once locked, a datum cannot be bumped anymore. Operation $locked(d)$ indicates if d is locked in the log. We write $d \in L$ when datum d is at some position in the log L. A log implies an ordering on the data items it contains. When d and d' both appear in L, $d <_L d'$ is true when the position of d is lower than the position of d', or they both occupy the same slot and $d < d'$, for some a priori total order ($<$) over the data items.

Variables. Algorithm 1 employs two types of shared objects at a process. First, for any two groups h and h' to which the local process belongs, Algorithm 1 uses a log $\text{LOG}_{h \cap h'}$ (line 2). Notice that, when $h = h'$, the log coincides with the log of the destination group h, i.e., LOG_h. Second, to agree on the final position of a message, Algorithm 1 also employs consensus objects (line 3). Consensus objects are both indexed by messages and group families. Given some message m and appropriate family $\mathfrak{f}$, Algorithm 1 calls $\text{CONS}_{m,\mathfrak{f}}$ (lines 20 and 21). Two processes call the same consensus object at line 21 only if both parameters match. Finally, to store the status of messages addressed to the local process, Algorithm 1 also employs a mapping PHASE (line 4). A message is initially in the `start` phase, then it moves to `pending` (line 15), `commit` (line 24), `stable` (line 33) and finally the `deliver` (line 37) phase. Phases are ordered according to this progression.

Algorithm 1 Solving atomic multicast with failure detector μ – code at process p.

```
1:  variables:
2:      (LOG_{h∩h'})_{h,h'∈G(p)}
3:      (CONS_{m,f})_{m∈M,f⊆G}
4:      PHASE[m] ← λm.start

5:  multicast(m) :=                                    // g = dst(m) ∧ g ∈ G(p)
6:      pre:  PHASE[m] = start
7:      eff:  LOG_g.append(m)

8:  pending(m) :=
9:      pre:  PHASE[m] = start
10:           m ∈ LOG_g
11:           ∀m' <_{LOG_g} m. PHASE[m'] ≥ commit
12:     eff:  for all h ∈ G(p) do
13:               i ← LOG_{g∩h}.append(m)
14:               LOG_g.append(m, h, i)
15:           PHASE[m] ← pending

16: commit(m) :=
17:     pre:  PHASE[m] = pending
18:           ∀h ∈ γ(g). (m, h, −) ∈ LOG_g
19:     eff:  let k = max{i : ∃(m, −, i) ∈ LOG_g},
20:           let f = {h : ∃f' ∈ F(p). g, h ∈ f' ∧ g ∩ h ≠ ∅}
21:           k ← CONS_{m,f}.propose(k)
22:           for all h ∈ G(p) do
23:               LOG_{g∩h}.bumpAndLock(m, k)
24:           PHASE[m] ← commit

25: stabilize(m, h) :=
26:     pre:  PHASE[m] = commit
27:           h ∈ G(p)
28:           ∀m' <_{LOG_{g∩h}} m. PHASE[m'] ≥ stable
29:     eff:  LOG_g.append(m, h)

30: stable(m) :=
31:     pre:  PHASE[m] = commit
32:           ∀h ∈ γ(g). (m, h) ∈ LOG_g
33:     eff:  PHASE[m] ← stable

34: deliver(m) :=
35:     pre:  PHASE[m] = stable
36:           ∀m' <_{LOG_{g∩h}} m. PHASE[m'] = deliver
37:     eff:  PHASE[m] ← deliver
```

Algorithmic details. We now detail Algorithm 1 and jointly argue about its correctness. For clarity, our argumentation is informal – the full proof appears in [37].

To multicast some message m to $g = dst(m)$, a process adds m to the log of its destination group (line 7). When $p \in g$ observes m in the log, p appends m to each $\text{LOG}_{g\cap h}$ with $p \in g \cap h$ (line 13). Then, p stores in the log of the destination group of m the slot occupied by m in $\text{LOG}_{g\cap h}$ (line 14). This moves m to the `pending` phase.

Similarly to Skeen's algorithm [5], a message is then bumped to the highest slot it occupies in the logs. This step is executed at lines 16-24. In detail, p first agrees with its peers on the highest position k occupied by m (lines 19-21). Observe here that only the processes

in g that share some cyclic family with p take part to this agreement (line 20). Then, for each group h in $\mathcal{G}(p)$, p bumps m to slot k in $\text{LOG}_{g \cap h}$ and locks it in this position (line 23). This moves m to the `commit` phase.

The next steps of Algorithm 1 compute the predecessors of message m. With more details, once m reaches the `stable` phase and is ready to be delivered, the messages that precede it in the logs at process p cannot change anymore.

If g does not belong to any cyclic family, stabilizing m is immediate: the precondition at line 32 is always vacuously true. In this case, m is delivered in an order consistent with the order it is added to the logs (line 28). This comes from the fact that when $\mathcal{F} = \varnothing$ ordering the messages reduces to a pairwise agreement between the processes.

When $\mathcal{F} \neq \varnothing$, stabilizing m is a bit more involved. Indeed, messages can be initially in cyclic positions, e.g., $C = m_1 <_{\text{LOG}_{g_1 \cap g_2}} m_2 <_{\text{LOG}_{g_2 \cap g_3}} m_3 <_{\text{LOG}_{g_3 \cap g_1}} m_1$, preventing them to be delivered. As in [5], bumping messages helps to resolve such a situation.

The bumping procedure is executed globally. A process must wait that the positions in the logs of a message are cycle-free before declaring it `stable`. Waiting can cease when the cyclic family is faulty (line 32). This is correct because messages are stabilized in the order of their positions in the logs (lines 25-29). Hence, if a cycle C exists initially in the positions, either (i) not all the messages in C are delivered, or (ii) the first message to get `stable` in C has no predecessors in C in the logs. In other words, for any two messages m and m' in C, if $m \mapsto m'$ then m is `stable` before m'.

A process indicates that message m with $g = dst(m)$ is stabilized in group h with a pair (m, h) in LOG_g (line 29). When this holds for all the groups h intersecting with g such that there exists a correct family $\mathfrak{f}$ with $f \in \mathcal{F}(p)$ and $g, h \in \mathfrak{f}$, m is declared `stable` at p (line 32). Once `stable`, a message m can be delivered (lines 34-37).

Algorithm 1 stabilizes then delivers messages according to their positions in the logs. To maintain progress, these positions must remain acyclic at every correct process. Furthermore, this should also happen globally when a cyclic family is correct. Both properties are ensured by the calls to consensus objects (line 21).

Implementing the shared objects. In each group g, consensus is solvable since μ provides $\Sigma_g \wedge \Omega_g$. This serves to implement all the objects $(\text{CONS}_{m,\mathfrak{f}})_{m,\mathfrak{f}}$ when $dst(m) = g$. Logs that are specific to a group, namely $(\text{LOG}_g)_{g \in \mathcal{G}}$, are also built atop consensus in g using a universal construction [27].

Failure detector μ does not offer the means to solve consensus in $g \cap h$. Hence we must rely on either g or h to build $\text{LOG}_{g \cap h}$. Minimality requires processes in a destination group to take steps only in the case a message is addressed to them. To achieve this, we have to slightly modify the universal construction for $\text{LOG}_{g \cap h}$, as detailed next.

First, we consider that this construction relies on an unbounded list of consensus objects.[2] Each consensus object in this list is contention-free fast [2]. This means that it is guarded by an adopt-commit object (AC) [19] before an actual consensus object (CONS) is called. Upon calling *propose*, AC is first used and if it fails, that is "adopt" is returned, CONS is called. Adopt-commit objects are implemented using $\Sigma_{g \cap h}$, while consensus objects are implemented atop some group, say g, using $\Sigma_g \wedge \Omega_g$. This modification ensures that when processes execute operations in the exact same order, only the adopt-commit objects are called. As a consequence, when no message is addressed to either g or h during a run, only the processes in $g \cap h$ executes steps to implement an operation of $\text{LOG}_{g \cap h}$.

[2] In the failure detector model, computability results can use any amount of shared objects.

Algorithm 2 Emulating $\Sigma_{\cap_{g \in G} g}$ – code at process p.

```
1: variables:
2:     (A_{g,x})_{g∈G, x⊆g.p∈x}
3:     (Q_g)_{g∈G} ← λg.{g}
4:     (qr_g)_{g∈G} ← λg.g
5: for all g ∈ G, x ⊆ g : p ∈ x do
6:     let m such that dst(m) = g ∧ payload(m) = p
7:     A_{g,x}.multicast(m)
8: when A_{g,x}.deliver(−)
9:     Q_g ← Q_g ∪ {x}
10: when query
11:     if p ∉ ⋂_{g∈G} g then
12:         return ⊥
13:     for all g ∈ G do
14:         qr_g ← choose argmax_{y∈Q_g} rank(y)
15:     return (⋃_{g∈G} qr_g) ∩ (⋂_{g∈G} g)
```

5 Necessity

Consider some environment $\mathfrak{E}$, a failure detector D and an algorithm A that uses D to solve atomic multicast in $\mathfrak{E}$. This section shows that D is stronger than μ in $\mathfrak{E}$. To this end, we first use the fact that atomic multicast solves consensus per group. Hence μ is stronger than $\wedge_{g \in \mathcal{G}} (\Omega_g \wedge \Sigma_g)$. §5.1 proves that D is stronger than $\Sigma_{g \cap h}$ for any two groups $g, h \in \mathcal{G}$. Further, in §5.2, we establish that D is stronger than γ. This last result is established when D is realistic. The remaining cases are discussed in §7.

5.1 Emulating $\Sigma_{g \cap h}$

Atomic multicast solves consensus in each destination group. This permits to emulate $\wedge_{g \in \mathcal{G}} \Sigma_g$. However, for two intersecting groups g and h, $\Sigma_g \wedge \Sigma_h$ is not strong enough to emulate $\Sigma_{g \cap h}$.[3] Hence, we must build the failure detector directly from the communication primitive. Algorithm 2 presents such a construction. This algorithm can be seen as an extension of the work of Bonnet and Raynal [6] to extract Σ_k when k-set agreement is solvable. Algorithm 2 emulates $\Sigma_{\cap_{g \in G} g}$, where $G \subseteq \mathcal{G}$ is a set of at most two intersecting destination groups.

At a process p, Algorithm 2 employs multiple instances of algorithm A. In detail, for every group $g \in G$ and subset x of g, if process p belongs to x, then p executes an instance $A_{g,x}$ (line 2). Variable Q_g stores the responsive subsets of g, that is the sets $x \subseteq g$ for which $A_{g,x}$ delivers a message. Initially, this variable is set to $\{g\}$.

Algorithm 2 uses the ranking function defined in [6]. For some set $x \subseteq \mathcal{P}$, function $rank(x)$ outputs the rank of x. Initially, all the sets have rank 0. Function $rank$ ensures a unique property: a set x is correct if and only if it ranks grows forever. To compute this function, processes keep track of each others by exchanging (asynchronously) "alive" messages. At a process p, the number of "alive" messages received so far from q defines the rank of q. The rank of a set is the lowest rank among all of its members.

[3] The two detectors may return forever non-intersecting quorums.

At the start of Algorithm 2, a process atomic multicasts its identity for every instance $A_{g,x}$ it is executing (line 7). When, $A_{g,x}$ delivers a process identity, x is added to variable Q_g (line 9). Thus, variable Q_g holds all the instances $A_{g,x}$ that progress successfully despite that $g \setminus x$ do not participate. From this set, Algorithm 2 computes the most responsive quorum using the ranking function (line 14). As stated in Theorem 2 below, these quorums must intersect at any two processes in $\cap_{g \in G} g$.

▶ **Theorem 2.** *Algorithm 2 implements $\Sigma_{\cap_{g \in G} g}$ in $\mathfrak{E}$.*

5.2 Emulating γ

Target systems. A process p is failure-prone in environment $\mathfrak{E}$ when for some failure pattern $F \in \mathfrak{E}$, $p \in \mathit{Faulty}(F)$. By extension, we say that $P \subseteq \mathcal{P}$ is failure-prone when for some $F \in \mathfrak{E}$, $P \subseteq \mathit{Faulty}(F)$. A cyclic family $\mathfrak{f}$ is failure-prone when one of its group intersections is failure-prone. Below, we consider that $\mathfrak{E}$ satisfies that if a process may fail, it may fail at any time (formally, $\forall F \in \mathfrak{E}. \forall p \in \mathit{Faulty}(F). \exists F' \in \mathfrak{E}. \forall t \in \mathbb{N}. \forall t' < t. F'(t') = F(t') \wedge F'(t) = F(t) \cup \{p\}$). We also restrict our attention to realistic failure detectors, that is they cannot guess the future [14].

Additional notions. Consider a cyclic family $\mathfrak{f}$. Two closed paths π and π' in $\mathit{cpaths}(\mathfrak{f})$ are equivalent, written $\pi \equiv \pi'$, when they visit the same edges in the intersection graph. A closed path π in $\mathit{cpaths}(\mathfrak{f})$ is oriented. The direction of π is given by $\mathit{dir}(\pi)$. It equals 1 when the path is clockwise, and -1 otherwise (for some canonical representation of the intersection graph). To illustrate these notions, consider family $\mathfrak{f}$ in Figure 1b. The sequence $\pi = g_3 g_1 g_2 g_3$ is a closed path in its intersection graph, with $|\pi| = 4$ and $\pi[0] = \pi[|\pi| - 1] = g_3$. The direction of this path is 1 since it is visiting clockwise the intersection graph of $\mathfrak{f}$ in the figure. Path π is equivalent to the path $\pi' = g_1 g_3 g_2 g_1$ which visits $\mathfrak{f}$ in the converse direction.

Construction. We emulate failure detector γ in Algorithm 3. For each closed path $\pi \in \mathit{cpaths}(\mathfrak{f})$ with $\pi[0] \cap \pi[1]$ failure-prone in $\mathfrak{E}$, Algorithm 3 maintains two variables: an instance A_π of the multicast algorithm A, and a flag $\mathit{failed}[\pi]$. Variable A_π is used to detect when a group intersection visited by π is faulty. It this happens, the flag $\mathit{failed}[\pi]$ is raised. When for every path $\pi \in \mathit{cpaths}(\mathfrak{f})$, some path equivalent to π is faulty, Algorithm 3 ceases returning the family $\mathfrak{f}$ (line 16).

In Algorithm 3, for every path $\pi \in \mathit{cpaths}(\mathfrak{f})$, the processes in $\pi[0] \cap \pi[1]$ multicast their identities to $\pi[0]$ using instance A_π (lines 4 and 5). In this instance of A, all the processes in $\mathfrak{f}$ but the intersection $\pi[0] \cap \pi[|\pi| - 2]$ participate (line 2). As the path is closed, this corresponds to the intersection with the last group preceding the first group in the path.

When $p \in \pi[i] \cap \pi[i+1]$ delivers a message $(-, i)$, it signals this information to the other members of the family (line 9). Then, p multicasts its identity to $\pi[i+1]$ (line 10). This mechanism is repeated until the antepenultimate group in the path is reached (line 8). When such a situation occurs, the flag $\mathit{failed}[\pi]$ is raised (line 12). This might also happen earlier when a message is received for some path π' equivalent to π and visiting $\mathfrak{f}$ in the converse direction (line 13).

Below, we claim that Algorithm 3 is a correct emulation of failure detector γ.

▶ **Theorem 3.** *Algorithm 3 implements γ in $\mathfrak{E}$.*

Algorithm 3 Emulating γ – code at process p.

```
1:  variables:
2:      (A_π)_π                              // ∀𝔣 ∈ 𝓕(p). ∀π ∈ cpaths(𝔣). p ∉ π[0] ∩ π[|π| − 2]
3:      failed[π] ← λπ.false

4:  for all A_π : p ∈ π[0] ∩ π[1] do
5:      A_π.multicast(p, 0) to π[0]

6:  signal(π, i) :=
7:      pre:  A_π.deliver(−, i)
8:            i < |π| − 2 ∧ p ∈ π[i + 1]
9:      eff:  send(π, i) to 𝔣
10:           A_π.multicast(p, i + 1) to π[i + 1]

11: update(π) :=
12:     pre:  ∃π′ ≡ π. rcv(π, j) ∧ ∨ j = |π| − 3
13:                                ∨ (rcv(π′, 0) ∧ π[j] = π′[0] ∧ dir(π) = −dir(π′))
14:     eff:  failed[π] ← true

15: when query
16:     return {𝔣 ∈ 𝓕(p) : ∃π ∈ cpaths(𝔣). ∀π′ ≡ π. failed[π′] = false}
```

6 Variations

This section explores two common variations of the atomic multicast problem. It shows that each variation has a weakest failure detector stronger than μ. The first variation requires messages to be ordered according to real time. This means that if m is delivered before m' is multicast, no process may deliver m' before m. In this case, we establish that the weakest failure detector must accurately detect the failure of a group intersection. The second variation demands each group to progress independently in the delivery of the messages. This property strengthens minimality because in a genuine solution a process may help others as soon as it has delivered a message. We show that the weakest failure detector for this variation permits to elect a leader in each group intersection.

6.1 Enforcing real-time order

Ordering primitives like atomic broadcast are widely used to construct dependable services [7]. The classical approach is to follow state-machine replication (SMR), a form of universal construction. In SMR, a service is defined by a deterministic state machine, and each replica maintains its own local copy of the machine. Commands accessing the service are funneled through the ordering primitive before being applied at each replica on the local copy.

SMR protocols must satisfy linearizability [28]. However, as observed in [3], the common definition of atomic multicast is not strong enough for this: if some command d is submitted after a command c get delivered, atomic multicast does not enforce c to be delivered before d, breaking linearizability. To sidestep this problem, a stricter variation must be used. Below, we define such a variation and characterize its weakest failure detector.

6.1.1 Definition

We write $m \rightsquigarrow m'$ when m is delivered in real-time before m' is multicast. Atomic multicast is *strict* when ordering is replaced with: (*Strict Ordering*) The transitive closure of $(\mapsto \cup \rightsquigarrow)$ is a strict partial order over $\mathcal{M}$. Strictness is free when there is a single destination group.

Algorithm 4 Emulating $1^{g \cap h}$ – code at process $p \in g \cup h$.

```
1:  variables:
2:      B ← if (p ∈ g \ h) then A_g else if (p ∈ h \ g) then A_h else ⊥  // A_g and A_h are distinct
    instances of A
3:      failed ← false

4:  if B ≠ ⊥ then
5:      B.multicast(p)
6:      wait until B.deliver(−)
7:      send(failed) to g ∪ h

8:  when rcv(failed)
9:      failed ← true

10: when query
11:     return failed
```

Indeed, if p delivers m before q broadcasts m', then necessarily $m \overset{p}{\mapsto} m'$. This explains why atomic broadcast does not mention such a requirement. In what follows, we prove that strict atomic multicast is harder than (vanilla) atomic multicast.

6.1.2 Weakest failure detector

Candidate. For some (non-empty) group of processes P, the *indicator failure detector* 1^P indicates if all the processes in P are faulty or not. In detail, this failure detector returns a boolean which ensures that:

(Accuracy) $\forall p \in \mathcal{P}. \forall t \in \mathbb{N}. 1^P(p,t) \Rightarrow P \subseteq F(t)$
(Completeness) $\forall p \in Correct. \forall t \in \mathbb{N}. P \subseteq F(t) \Rightarrow \exists \tau \in \mathbb{N}. \forall t' \geq \tau. 1^P(p,t')$

For simplicity, we write $1^{g \cap h}$ the indicator failure detector restricted to the processes in $g \cup h$ (that is, the failure detector $1^{g \cap h}_{g \cup h}$). This failure detector informs the processes outside $g \cap h$ when the intersection is faulty. Notice that for the processes in the intersection, $1^{g \cap h}$ does not provide any useful information. This comes from the fact that simply returning always *true* is a valid implementation at these processes.

Our candidate failure detector is $\mu \wedge (\wedge_{g,h \in \mathcal{G}} 1^{g \cap h})$. One can establish that $\wedge_{g,h \in \mathcal{G}} 1^{g \cap h}$ is stronger than γ (see Proposition 4 below). As a consequence, this failure detector can be rewritten as $(\wedge_{g,h \in \mathcal{G}} \Sigma_{g \cap h} \wedge 1^{g \cap h}) \wedge (\wedge_{g \in \mathcal{G}} \Omega_g)$.

▶ **Proposition 4.** $\wedge_{g,h \in \mathcal{G}} 1^{g \cap h} \leq \gamma$

Necessity. An algorithm to construct $1^{g \cap h}$ is presented in Algorithm 4. It relies on an implementation A of strict atomic multicast that makes use internally of some failure detector D. Proposition 5 establishes the correctness of such a construction.

▶ **Proposition 5.** *Algorithm 4 implements* $1^{g \cap h}$.

Sufficiency. The solution to strict atomic multicast is almost identical to Algorithm 1. The only difference is at line 32 when a message moves to the `stable` phase. Here, for every destination group h with $h \cap g \neq \varnothing$, a process waits either that $1^{g \cap h}$ returns *true*, or that a tuple (m, h) appears in LOG_g. From Proposition 4, we know that the indicator failure detector $1^{g \cap h}$ provides a better information than γ regarding the correctness of $g \cap h$. As a consequence, the modified algorithm solves (group sequential) atomic multicast.

Now, to see why such a solution is strict, consider two messages m and m' that are delivered in a run, with $g = dst(m)$ and $h = dst(m')$. We observe that when $m' \rightsquigarrow m$ or $m' \mapsto m$, m' is stable before m, from which we deduce that strict ordering holds.

With more details, in the former case ($m' \rightsquigarrow m$), this comes from the fact that to be delivered a message must be `stable` first (line 35). In the later ($m' \mapsto m$), when message m is `stable` at some process p, p must wait a message (m, h) in LOG_g, or that $1^{g \cap h}$ returns *true*. If (m, h) is in LOG_g, then line 29 was called before by some process q. Because both messages are delivered and $m' \mapsto m$, m' must precedes m in $\text{LOG}_{g \cap h}$. Thus the precondition at line 28 enforces that m' is `stable` at q, as required. Now, if the indicator returns *true* at p, $m' \mapsto m$ tells us that a process delivers m' before m and this must happen before $g \cap h$ fails.

6.2 Improving parallelism

As motivated in the Introduction, genuine solutions to atomic multicast are appealing from a performance perspective. Indeed, if messages are addressed to disjoint destination groups in a run, they are processed in parallel by such groups. However, when contention occurs, a message may wait for a chain of messages to be delivered first. This chain can span outside of the destination group, creating a delay that harms performance and reduces parallelism [17, 1]. In this section, we explore a stronger form of genuineness, where groups are able to deliver messages independently. We prove that, similarly to the strict variation, this requirement demands more synchrony than μ from the underlying system.

6.2.1 Definition

As standard, a run R is fair for some correct process p when p executes an unbounded amount of steps in R. By extension, R is fair for $P \subseteq \mathit{Correct}(R)$, or for short P-fair, when it is fair for every p in P. If P is exactly the set of correct processes, we simply say that R is fair.

(Group Parallelism) Consider a message m and a run R. Note $P = \mathit{Correct}(R) \cap dst(m)$. If m is delivered by a process, or atomic multicast by a correct process in R, and R is P-fair, then every process in P delivers m in R.

Group parallelism bears similarity with x-obstruction freedom [38], in the sense that the system must progress when a small enough group of processes is isolated. A protocol is said *strongly genuine* when it satisfy both the minimality and the group parallelism properties.

6.2.2 About the weakest failure detector

Below, we establish that $(\wedge_{g,h \in \mathcal{G}} \Omega_{g \cap h})$ is necessary. It follows that the weakest failure detector for this variation is at least $\mu \wedge (\wedge_{g,h \in \mathcal{G}} \Omega_{g \cap h})$.

Emulating $\wedge_{g,h \in \mathcal{G}} \Omega_{g \cap h}$. Consider some algorithm A that solves strongly genuine atomic multicast with failure detector D. Using both A and D, each process may emulate $\Omega_{g \cap h}$, for some intersecting groups $g, h \in \mathcal{G}$. The emulation follows the general schema of CHT [8]. We sketch the key steps below. The full proof appears in [37].

Each process constructs a directed acyclic graph G by sampling the failure detector D and exchanging these samples with other processes. A path π in G induces multiple runs of A that each process locally simulates. A run starts from some initial configuration. In our context, the configurations $\mathfrak{I} = \{I_1, \ldots, I_{n \geq 2}\}$ of interest satisfy (i) the processes outside

$g \cap h$ do not atomic multicast any message, and (ii) the processes in $g \cap h$ multicast a single message to either g or h. For some configuration $I_i \in \mathfrak{I}$, the schedules of the simulated runs starting from I_i are stored in a simulation tree Υ_i. There exists an edge (S, S') when starting from configuration $S(I_i)$, one may apply a step $s = (p, m, d)$ for some process p, message m transiting in $S(I_i)$ and sample d of D such that $S' = S \cdot s$.

Every time new samples are received, the forest of the simulation trees $(\Upsilon_i)_i$ is updated. At each such iteration, the schedules in Υ_i are tagged using the following valency function: S is tagged with g (respectively, h) if for some successor S' of S in Υ_i a process in $g \cap h$ delivers first a message addressed to g (resp. to h) in configuration $S'(I_i)$. A tagged schedule is *univalent* when it has a single tag, and *bivalent* otherwise.

As the run progresses, each root of a simulation tree has eventually a stable set of tags. If the root of Υ_i is g-valent, the root of Υ_j is h-valent and they are adjacent, i.e., all the processes but some $p \in g \cap h$ are in the same state in I_i and I_j, then p must be correct. Otherwise, there exists a bivalent root of some tree Υ_i such that for g (respectively, h) a correct process multicasts a message to g (resp., h) in I_i. In this case, similarly to [8], there exists a decision gadget in the simulation tree Υ_i. This gadget is a sub-tree of the form (S, S', S''), with S bivalent, and S' g-valent and S'' h-valent (or vice-versa). Using the group parallelism property of A, we may then show that necessarily the deciding process in the gadget, that is the process taking a step toward either S' or S'' is correct and belongs to the intersection $g \cap h$.

Solution when $\mathcal{F} = \varnothing$. In this case, Algorithm 1 just works. To attain strong genuineness, each log object $\text{LOG}_{g \cap h}$ is implemented with $\Sigma_{g \cap h} \wedge \Omega_{g \cap h}$ through standard universal construction mechanisms. When $\mathcal{F} = \varnothing$, $\mu \wedge (\wedge_{g,h \in \mathcal{G}} \Omega_{g \cap h})$ is thus the weakest failure detector. The case $\mathcal{F} \neq \varnothing$ is discussed in the next section.

7 Discussion

Several definitions for atomic multicast appear in literature (see, e.g., [12, 26] for a survey). Some papers consider a variation of atomic multicast in which the ordering property is replaced with: (*Pairwise Ordering*) If p delivers m then m', every process q that delivers m' has delivered m before. Under this definition, cycles in the delivery relation ($\mapsto$) across more than two groups are not taken into account. This is computably equivalent to $\mathcal{F} = \varnothing$. Hence the weakest failure detector for this variation is $(\wedge_{g,h \in \mathcal{G}} \Sigma_{g \cap h}) \wedge (\wedge_{g \in \mathcal{G}} \Omega_g)$.

In [25], the authors show that failure detectors of the class $\mathcal{U}_2$ are too weak to solve the pairwise ordering variation. These detectors can be wrong about (at least) two processes. In detail, the class $\mathcal{U}_k$ are all the failure detectors D that are k-unreliable, that is they cannot distinguish any pair of failure patterns F and F', as long as the faulty processes in F and F' are members of a subset W of size k (the "wrong" subset). The result in [25] is a corner case of the necessity of $\Sigma_{g \cap h}$ when $g \cap h = \{p, q\}$ and both processes are failure-prone in $\mathfrak{E}$. Indeed, $\Sigma_{\{p,q\}} \notin \mathcal{U}_2$. To see this, observe that if q is faulty and p correct, then $\{p\}$ is eventually the output of $\Sigma_{\{p,q\}}$ at p. A symmetrical argument holds for process q in runs where q is correct and p faulty. In the class $\mathcal{U}_2$, such values can be output in runs where both processes are correct, contradicting the intersection property of $\Sigma_{\{p,q\}}$.

Most atomic multicast protocols [30, 17, 20, 10, 31, 29, 13, 33] sidestep the impossibility result in [25] by considering that destination groups are decomposable into a set of disjoint groups, each of these behaving as a logically correct entity. This means that there exists a partitioning $\mathfrak{P}(\mathcal{G}) \subseteq 2^{\mathcal{P}}$ satisfying that (i) for every destination group $g \in \mathcal{G}$, there exists

$(g_i)_i \subseteq \mathfrak{P}(\mathcal{G})$ with $g = \cup_i g_i$, (ii) each $g \in \mathfrak{P}(\mathcal{G})$ is correct, and (iii) for any two g, h in $\mathfrak{P}(\mathcal{G})$, $g \cap h$ is empty. Since $\wedge_{g \in \mathfrak{P}(\mathcal{G})} (\Sigma_g \wedge \Omega_g) \succeq \mu$, we observe that solving the problem over $\mathfrak{P}(\mathcal{G})$ is always as difficult as over $\mathcal{G}$. It can also be more demanding in certain cases, e.g., if two groups intersect on a single process p, then p must be reliable. In Figure 1, this happens with process p_2. In contrast, to these prior solutions, Algorithm 1 tolerates any number of failures. This is also the case of [34] which relies on a perfect failure detector.

Regarding strongly genuine atomic multicast, §6.2 establishes that $\mu \wedge (\wedge_{g,h \in \mathcal{G}} \Omega_{g \cap h})$ is the weakest when $\mathcal{F} = \varnothing$. The case $\mathcal{F} \neq \varnothing$ is a bit more intricate. First of all, we may observe that in this case the problem is failure-free solvable: given a spanning tree T of the intersection graph of $\mathcal{G}$, we can deliver the messages according to the order $<_T$, that is, if m is addressed to g intersecting with $h, h', \ldots$ with $h <_T h' <_T \ldots$, then $g \cap h$ delivers first m, followed by $g \cap h'$, etc.[4] A failure-prone solution would apply the same logic. This is achievable using $\mu \wedge (\wedge_{g,h \in \mathcal{G}} \Omega_{g \cap h}) \wedge (\wedge_{g,h \in \mathcal{F}} 1^{g \cap h})$, where $g \in \mathcal{F}$ holds when for some family $\mathfrak{f} \in \mathcal{F}$, we have $g \in \mathfrak{f}$. We conjecture that this failure detector is actually the weakest.

8 Conclusion

This paper presents the first solution to genuine atomic multicast that tolerates arbitrary failures without using system-wide perfect failure detection. It also introduces two new classes of failure detectors: (γ) which tracks when a cyclic family of destination groups is faulty, and $(1^{g \cap h})$ that indicates when the group intersection $g \cap h$ is faulty. Building upon these new abstractions, we identify the weakest failure detector for genuine atomic multicast and also for several key variations of this problem. Our results offer a fresh perspective on the solvability of genuine atomic multicast in crash-prone systems. In particular, they question the common assumption of partitioning the destination groups. This opens an interesting avenue for future research on the design of fault-tolerant atomic multicast protocols.

References

1 Tarek Ahmed-Nacer, Pierre Sutra, and Denis Conan. The convoy effect in atomic multicast. In *35th IEEE Symposium on Reliable Distributed Systems Workshops, SRDS 2016 Workshop, Budapest, Hungary, September 26, 2016*, pages 67–72. IEEE Computer Society, 2016. doi:10.1109/SRDSW.2016.22.

2 Hagit Attiya, Rachid Guerraoui, and Petr Kouznetsov. Computing with reads and writes in the absence of step contention. In Pierre Fraigniaud, editor, *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings*, volume 3724 of *Lecture Notes in Computer Science*, pages 122–136. Springer, 2005. doi:10.1007/11561927_11.

3 Carlos Eduardo Benevides Bezerra, Fernando Pedone, and Robbert van Renesse. Scalable state-machine replication. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 331–342. IEEE Computer Society, 2014. doi:10.1109/DSN.2014.41.

4 Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, August 1991. doi:10.1145/128738.128742.

5 Kenneth P. Birman and Thomas A. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computers Systems*, 5(1):47–76, January 1987. doi:10.1145/7351.7478.

[4] Strictly speaking, a spanning tree is required per connected component of the intersection graph.

6 François Bonnet and Michel Raynal. Looking for the weakest failure detector for k-set agreement in message-passing systems: Is π_k the end of the road? In *Stabilization, Safety, and Security of Distributed Systems, 11th International Symposium, SSS 2009, Lyon, France, November 3-6, 2009. Proceedings*, pages 149–164, 2009. doi:10.1007/978-3-642-05118-0_11.

7 Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In Indranil Gupta and Roger Wattenhofer, editors, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007*, pages 398–407. ACM, 2007. doi:10.1145/1281100.1281103.

8 Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996. doi:10.1145/234533.234549.

9 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996. doi:10.1145/226643.226647.

10 Paulo R. Coelho, Nicolas Schiper, and Fernando Pedone. Fast atomic multicast. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*, pages 37–48. IEEE Computer Society, 2017. doi:10.1109/DSN.2017.15.

11 James A. Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. In Gernot Heiser and Wilson C. Hsieh, editors, *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 223–235. USENIX Association, 2012. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/cowling.

12 Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, December 2004. doi:10.1145/1041680.1041682.

13 Carole Delporte-Gallet and Hugues Fauconnier. Fault-tolerant genuine atomic multicast to multiple groups. In Franck Butelle, editor, *Procedings of the 4th International Conference on Principles of Distributed Systems, OPODIS 2000, Paris, France, December 20-22, 2000*, Studia Informatica Universalis, pages 107–122. Suger, Saint-Denis, rue Catulienne, France, 2000.

14 Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. A realistic look at failure detectors. In *2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings*, pages 345–353. IEEE Computer Society, 2002. doi:10.1109/DSN.2002.1028919.

15 Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004*, pages 338–346, 2004. doi:10.1145/1011767.1011818.

16 Swan Dubois, Rachid Guerraoui, Petr Kuznetsov, Franck Petit, and Pierre Sens. The weakest failure detector for eventual consistency. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 375–384, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2767386.2767404.

17 Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. Efficient replication via timestamp stability. In Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar, editors, *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 178–193. ACM, 2021. doi:10.1145/3447786.3456236.

18 Felix C. Freiling, Rachid Guerraoui, and Petr Kuznetsov. The failure detector abstraction. *ACM Comput. Surv.*, 43(2), February 2011. doi:10.1145/1883612.1883616.

19 Eli Gafni. Round-by-round fault detectors (extended abstract): Unifying synchrony and asynchrony. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '98, pages 143–152, New York, NY, USA, 1998. ACM. doi:10.1145/277697.277724.

20 Alexey Gotsman, Anatole Lefort, and Gregory V. Chockler. White-box atomic multicast. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*, pages 176–187. IEEE, 2019. doi:10.1109/DSN.2019.00030.

21 R. Guerraoui and A. Schiper. Total order multicast to multiple groups. In *Proceedings of 17th International Conference on Distributed Computing Systems*, pages 578–585, 1997. doi:10.1109/ICDCS.1997.603426.

22 Rachid Guerraoui, Vassos Hadzilacos, Petr Kuznetsov, and Sam Toueg. The weakest failure detectors to solve quittable consensus and nonblocking atomic commit. *SIAM J. Comput.*, 41(6):1343–1379, 2012. doi:10.1137/070698877.

23 Rachid Guerraoui, Maurice Herlihy, Petr Kouznetsov, Nancy Lynch, and Calvin Newport. On the weakest failure detector ever. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 235–243, New York, NY, USA, 2007. ACM. doi:10.1145/1281100.1281135.

24 Rachid Guerraoui and Michel Raynal. The alpha of indulgent consensus. *Comput. J.*, 50(1):53–67, 2007. doi:10.1093/comjnl/bxl046.

25 Rachid Guerraoui and André Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theor. Comput. Sci.*, 254(1-2):297–316, 2001. doi:10.1016/S0304-3975(99)00161-9.

26 Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, 1994.

27 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991. doi:10.1145/114005.102808.

28 Maurice Herlihy and Jeannette Wing. Linearizability: a correcteness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990. doi:10.1145/78969.78972.

29 Udo Fritzke Jr., Philippe Ingels, Achour Mostéfaoui, and Michel Raynal. Consensus-based fault-tolerant total order multicast. *IEEE Trans. Parallel Distributed Syst.*, 12(2):147–156, 2001. doi:10.1109/71.910870.

30 Long Hoang Le, Mojtaba Eslahi-Kelorazi, Paulo R. Coelho, and Fernando Pedone. Ramcast: Rdma-based atomic multicast. In Kaiwen Zhang, Abdelouahed Gherbi, Nalini Venkatasubramanian, and Luís Veiga, editors, *Middleware '21: 22nd International Middleware Conference, Québec City, Canada, December 6 - 10, 2021*, pages 172–184. ACM, 2021. doi:10.1145/3464298.3493393.

31 Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. Multi-ring paxos. In Robert S. Swarz, Philip Koopman, and Michel Cukier, editors, *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2012, Boston, MA, USA, June 25-28, 2012*, pages 1–12. IEEE Computer Society, 2012. doi:10.1109/DSN.2012.6263916.

32 Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 517–532. USENIX Association, 2016. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/mu.

33 Luís E. T. Rodrigues, Rachid Guerraoui, and André Schiper. Scalable atomic multicast. In *Proceedings of the International Conference On Computer Communications and Networks (IC-CCN 1998), October 12-15, 1998, Lafayette, Louisiana, USA*, pages 840–847. IEEE Computer Society, 1998. doi:10.1109/ICCCN.1998.998851.

34 Nicolas Schiper and Fernando Pedone. Solving atomic multicast when groups crash. In Theodore P. Baker, Alain Bui, and Sébastien Tixeuil, editors, *Principles of Distributed Systems, 12th International Conference, OPODIS 2008, Luxor, Egypt, December 15-18, 2008. Proceedings*, volume 5401 of *Lecture Notes in Computer Science*, pages 481–495. Springer, 2008. doi:10.1007/978-3-540-92221-6_30.

35 Nicolas Schiper, Pierre Sutra, and Fernando Pedone. Genuine versus non-genuine atomic multicast protocols for wide area networks: An empirical study. In *28th IEEE Symposium on Reliable Distributed Systems (SRDS 2009), Niagara Falls, New York, USA, September 27-30, 2009*, pages 166–175. IEEE Computer Society, 2009. doi:10.1109/SRDS.2009.12.

36 Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-store: Genuine partial replication in wide area networks. In *29th IEEE Symposium on Reliable Distributed Systems (SRDS 2010), New Delhi, Punjab, India, October 31 - November 3, 2010*, pages 214–224. IEEE Computer Society, 2010. doi:10.1109/SRDS.2010.32.

37 Pierre Sutra. The weakest failure detector for genuine atomic multicast (extended version), 2022. doi:10.48550/ARXIV.2208.07650.

38 Gadi Taubenfeld. Contention-sensitive data structures and algorithms. *Theoretical Computer Science*, 677:41–55, 2017. doi:10.1016/j.tcs.2017.03.017.

On Implementing SWMR Registers from SWSR Registers in Systems with Byzantine Failures

Xing Hu
Department of Computer Science, University of Toronto, Canada

Sam Toueg
Department of Computer Science, University of Toronto, Canada

Abstract

The implementation of registers from (potentially) weaker registers is a classical problem in the theory of distributed computing. Since Lamport's pioneering work [14], this problem has been extensively studied in the context of asynchronous processes with crash failures. In this paper, we investigate this problem in the context of Byzantine process failures, with and without process signatures. In particular, we first show a strong impossibility result, namely, that there is no wait-free linearizable implementation of a 1-writer n-reader register from atomic 1-writer $(n-1)$-reader registers. In fact, this impossibility result holds even if *all the processes except the writer* are given atomic 1-writer n-reader registers, and even if we assume that the writer can only crash and *at most one* reader is subject to Byzantine failures. In light of this impossibility result, we give two register implementations. The first one implements a 1-writer n-reader register from atomic 1-writer 1-reader registers. This implementation is linearizable (under any combination of Byzantine process failures), but it is wait-free only under the assumption that the writer is correct or no reader is Byzantine – thus matching the impossibility result. The second implementation assumes process signatures; it is wait-free and linearizable under any number and combination of Byzantine process failures.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Distributed algorithms

Keywords and phrases distributed computing, concurrency, linearizability, shared registers

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.36

Related Version *Full Version*: https://arxiv.org/abs/2207.01470

Funding This work was partially funded by the Natural Sciences and Engineering Research Council of Canada (Grant number: RGPIN-2014-05296).

Acknowledgements We thank Vassos Hadzilacos for his helpful comments on this paper.

1 Introduction

We consider the basic problem of implementing a single-writer *multi*-reader register from atomic single-writer *single*-reader registers in a system where processes are subject to *Byzantine failures*. In particular, (1) we give an implementation that works under some failure assumptions, and (2) we prove a matching impossibility result for the case when these assumptions do not hold. We also consider systems where processes can use unforgeable signatures, and give an implementation that works for any number of faulty processes. We now describe our motivation and results in detail.

1.1 Motivation

Implementing shared registers from weaker primitives is a fundamental problem that has been thoroughly studied in distributed computing [3, 4, 5, 8, 12, 14, 16, 17, 18, 19, 20, 21, 22].

© Xing Hu and Sam Toueg;
licensed under Creative Commons License CC-BY 4.0

In particular, it is well-known that in systems where processes are subject to *crash* failures, it is possible to implement a wait-free linearizable m-writer n-reader register (henceforth denoted $[m, n]$-register) from atomic 1-writer 1-reader registers (denoted $[1, 1]$-registers).

In this paper, we consider the problem of implementing *multi*-reader registers from *single*-reader registers in systems where processes are subject to *Byzantine* failures. In particular, we consider the following basic questions:

- Is there a wait-free linearizable implementation of a $[1, n]$-register from atomic $[1, 1]$-registers in systems with Byzantine processes?
- If so, under which assumption(s) such an implementation exist?

The above questions are also motivated by the growing interest in shared-memory or hybrid systems where processes are subject to Byzantine failures. For example, Cohen and Keidar [6] give f-resilient implementations of several objects (namely, *reliable broadcast*, *atomic snapshot*, and *asset transfer* objects) using atomic $[1, n]$-registers in systems with Byzantine failures where at most $f < n/2$ processes are faulty. As another example, Aguilera *et al.* use atomic $[1, n]$-registers to solve some agreement problems in hybrid systems with Byzantine process failures [1]. Moreover, Mostéfaoui *et al.* [15] prove that, in *message-passing* systems with Byzantine process failures, there is a f-resilient linearizable implementation of a $[1, n]$-register if and only if at most $f < n/3$ processes are faulty.

1.2 Description of the results

To simplify the exposition of our results, we first state them in terms of two process groups: *correct* processes that do not fail and *faulty* ones. We show that in a system with Byzantine failures the following impossibility and possibility results hold. For all $n \geq 3$:

(A) If the writer *and* some reader (even if only one of them) can be faulty, then there is no wait-free linearizable implementation of a $[1, n]$-register from atomic $[1, n-1]$-registers.

(B) If the writer *or* some readers (any number of them), but *not both*, can be faulty, then there is a wait-free linearizable implementation of a $[1, n]$-register from atomic $[1, 1]$-registers.

The case $n = 2$ is special: we give a wait-free linearizable implementation of a $[1, 2]$-register from atomic $[1, 1]$-registers that works even if the writer *and* readers can be faulty.

This simple version of the results, however, leaves several questions open. Intuitively, this is because the above results do not distinguish between the different types of faulty processes (recall that, by definition, Byzantine failures encompass all the possible failure behaviours, from simple crash to "malicious" behaviour). For example we may ask: what happens if we can assume that some processes (say the writer) are subject to crash failures *only*, while some other processes (say the readers) can fail in "malicious" ways? Is a wait-free linearizable implementation of a $[1, n]$-register from atomic $[1, 1]$-registers now possible?

Note also that the above results consider linearizability and wait-freedom (intuitively, "safety" and "liveness") as an *indivisible* requirement of a register implementation. But it can be useful to consider each requirement separately. For example, what happens if we want to implement a $[1, n]$-register with the following properties: (1) it is *always* safe (i.e., linearizable) and (2) it may lose its liveness (i.e., it may lose its wait-freedom by "blocking" some read or write operations) *only if* some specific "pattern/types" of failures occur?

To answer such questions, we now consider linearizability and wait-freedom separately, and we partition processes into *three* separate groups: (1) those that do not fail, called *correct* processes, (2) those that fail *only* by crashing, and (3) those that fail in any other way, called *malicious* processes. In systems with a mix of such processes, we prove the following:

(1) For all $n \geq 3$, there is *no* wait-free linearizable implementation I_n of a $[1, n]$-register from atomic $[1, n-1]$-registers, even if we assume that the writer can only crash and *at most one* of the readers can be malicious.

In fact, we show that this impossibility result holds even if *all* the processes except the writer are given atomic $[1, n]$-registers that all processes can read; so the writer is the *only* process that does not have an atomic $[1, n]$-register.

(2) For all $n \geq 3$, there is an implementation I_n of a $[1, n]$-register from atomic $[1, 1]$-registers such that:
- I_n is linearizable, and
- I_n is wait-free if the writer is correct or no reader is malicious.[1]

Note that this implementation guarantees *linearizability*, no matter which processes fail and how they fail (even if most processes are malicious). However, it guarantees *wait-freedom* only if the writer is correct or no reader is malicious.[2] So if the *readers* are subject to crash failures only, the implementation is wait-free *even if the writer is malicious*.
Note that the above impossibility and matching possibility results **(1)** and **(2)** imply the simpler results **(A)** and **(B)** that we stated earlier for processes that are (coarsely) characterized as either correct or faulty.

We also consider the problem of implementing a $[1, n]$-register from atomic $[1, 1]$-registers in systems where processes are subject to Byzantine failures, but they can use *unforgeable signatures*. In sharp contrast to the above results, we show that in such systems there is an implementation of $[1, n]$-register from atomic $[1, 1]$-registers that is linearizable *and* wait-free *no matter how many processes fail and how they fail*.

2 Result techniques

The techniques that we used to obtain our possibility and impossibility results (for the "no signatures" case) are also a significant contribution of this paper.

To prove the impossibility result **(1)**, one cannot use a standard partitioning argument: all the processes except the writer are given atomic $[1, n]$-registers that all processes can read, and the writer is given a $[1, n-1]$-register that all the readers except one can read; thus it is clear that the system cannot be partitioned.

So to prove this result we use an interesting *reductio ad absurdum* technique. Starting from an alleged implementation of $[1, n]$-register from $[1, n-1]$-registers, we consider a run where the implemented register is initialized to 0, the writer completes a write of 1, and then a reader reads 1. By leveraging the facts that: (1) in each step the writer can read or write only $[1, n-1]$-registers, (2) the writer may crash, (3) one of the readers may be malicious, (4) and there are at least 3 readers, we are able to successively remove every read or write step of the writer (one by one, starting from its last write operation) in a way that maintains the property that some correct reader reads 1 and at most one process in the run is malicious. As we successively remove the steps of the writer, the identity of the process that reads 1, and the identity of the process that may be malicious, keep changing. By continuing this process, we end up with a run in which the writer takes no steps, and yet a correct reader reads 1.

[1] That is, in every run of I_n where the writer is correct or no reader is malicious, correct processes complete all their operations.
[2] In fact it is slightly stronger than this: write operations are unconditionally "wait-free", only read operations may block if the condition is not met.

Note that this proof is reminiscent of the impossibility proof for the “Two generals’ Problem” in message-passing systems [7]. In that proof, one leverages the possibility of message losses to successively remove one message at a time. The proof given here is much more elaborate because it leverages the subtle interaction between crash *and* malicious failures that may occur at different processes.

For the matching possibility result **(2)**, we solve the problem of implementing a $[1, n]$-register from $[1, 1]$-registers with a *recursive* algorithm: intuitively, we first give an algorithm to implement a $[1, n]$-register using $[1, n-1]$-registers, rather than only $[1, 1]$-registers, and then recurse till $n = 2$. We do so because the recursive step of implementing a $[1, n]$-register using $[1, n-1]$-registers, is significantly easier than implementing a $[1, n]$-register using only $[1, 1]$-registers. This is explained in more detail in Section 5.1.

3 Model Sketch

We consider systems with asynchronous processes that communicate via single-writer registers and are subject to Byzantine failures. Recall that a single-writer n-reader register is denoted as a $[1, n]$-register; the n readers are distinct from the writer.

3.1 Process failures

A process that is subject to Byzantine failures can behave arbitrarily. In particular, it may deviate from the algorithm it is supposed to execute, or just stop this execution prematurely, i.e., crash. To distinguish between these two types of failures, we partition processes as follows:

- Processes that do not fail, i.e., *correct* processes.
- Processes that fail, i.e., *faulty* processes. Faulty processes are divided into two groups:
  - processes that just *crash*, and
  - the remaining processes, which we call *malicious*.

3.2 Atomic and implemented registers

A register is *atomic* if its read and write operations are *instantaneous* (i.e., indivisible); each read must return the value of the last write that precedes it, or the initial value of the register if no such write exists. Roughly speaking, the *implementation* of a register from a set of “base” registers is given by read/write procedures that each process can execute to read/write the implemented register; these procedures can access the given base registers (which, intuitively, may be less “powerful” than the implemented register). So each operation on an implemented register *spans an interval* that starts with an *invocation* (a procedure call) and completes with a corresponding *response* (a value returned by the procedure).

A register implementation is *wait-free* [2, 9, 13] if it guarantees that every operation invoked by a correct process completes with a response in a finite number of steps.

3.3 Linearizability of register implementations

Unless we explicitly state otherwise, all the register implementations that we consider are *linearizable* [10]. Intuitively, linearizability requires that every operation on an implemented object appears as if it took effect instantaneously at some point (the “linearization point”) in its execution interval.

As noted by [6, 15], however, the precise definition of linearizability depends on whether processes can only crash, or they can also fail in a “Byzantine way”. We now explain this for *register* implementations.

In systems with only crash failures. It is well-known that a *single-writer multi-reader* register implementation is linearizable if and only if it satisfies two simple properties. To define these properties precisely, we first define what it means for two operations to be concurrent or for one to precede the other.

▶ **Definition 1.** *Let o and o' be any two operations.*

- o precedes o' *if the response of o occurs before the invocation of o'.*
- o is concurrent with o' *if neither precedes the other.*

We say that a write operation W *immediately precedes* a read operation R if W precedes R, and there is no write operation W′ such that W precedes W′ and W′ precedes R.

Let v_0 be the *initial value* of the implemented register, and v_k be the value written by the k-th write operation of the writer w (this is well-defined because we make the standard assumption that each process applies operations sequentially).

▶ **Definition 2** (Register Linearizability). *In a system with crash failures, an implementation of a $[1, n]$-register is* linearizable *if and only if it satisfies the following two properties:*

- ***Property 1** [Reading a "current" value] If a read operation R returns the value v then:*
  - *there is a write v operation that immediately precedes R or is concurrent with R, or*
  - *$v = v_0$ and no write operation precedes R.*
- ***Property 2** [No "new-old" inversion] If two read operations R and R′ return values v_k and $v_{k'}$, respectively, and R precedes R′, then $k \leq k'$.*

In systems with Byzantine failures. The above definitions do not quite work for systems with Byzantine failures. For example, it is not clear what it means for a writer w of an implemented register to "write a value v" if w is malicious, i.e., if w *deviates* from the write procedure that it is supposed to execute; similarly, if a reader r is malicious it is not clear what it means for r to "read a value v". The definition of linearizability for systems with Byzantine failures avoids the above issues by restricting the linearization requirements to processes that are *not* malicious. More precisely:

▶ **Definition 3** (Register Linearizability). *In a system with Byzantine process failures, an implementation of a $[1, n]$-register is* linearizable *if and only if the following holds. If the writer is not malicious, then:*

- ***Property 1** [Reading a "current" value] If a read operation R by a process that is not malicious returns the value v then:*
  - *there is a write v operation that immediately precedes R or is concurrent with R, or*
  - *$v = v_0$ and no write operation precedes R.*
- ***Property 2** [No "new-old" inversion] If two read operations R and R′ by processes that are not malicious return values v_k and $v_{k'}$, respectively, and R precedes R′, then $k \leq k'$.*

Note that if the writer is correct or only crashes, then readers that are correct or only crash are required to read "current" values and also avoid "new-old" inversions. So in systems where faulty processes can only crash, Definition 3 reduces to Definition 2.

Cohen and Keidar were the first to define linearizability for *arbitrary* objects in systems with Byzantine failures [6], and their definition generalizes the definition of linearizability for $[1, n]$-*registers* given by Mostéfaoui *et al.* in [15]. Definition 3 is consistent with both.

We now describe the results of this paper. Because of space limitations, some of the proofs are omitted here; they can be found in [11].

4 Impossibility result

We now prove that in a system with $n+1$ Byzantine processes, if the writer *and* one of the n readers can be faulty, then there is no wait-free linearizable implementation of a $[1,n]$-register from atomic $[1,n-1]$-registers. In fact, by dividing faulty processes into those that can only crash and those that can be malicious (as defined in Section 3), we show the following stronger result.

▶ **Theorem 4.** *For all $n \geq 3$, there is* no *wait-free linearizable implementation of a $[1,n]$-register from atomic $[1,n-1]$-registers in a system with $n+1$ processes that are subject to Byzantine failures. This holds even if we assume that the writer of the implemented $[1,n]$-register can only crash and at most one reader can be malicious.*

Proof. Let $n \geq 3$. Suppose, for contradiction, that there is a wait-free linearizable implementation $\mathcal{I}$ of a $[1,n]$-register $\mathbf{R}$ from atomic $[1,n-1]$-registers, in a system where the writer w of $\mathbf{R}$ can crash and one of the n readers of $\mathbf{R}$ can be malicious.

We now construct a sequence of executions of $\mathcal{I}$ that leads to a contradiction. In all these executions, the initial value of the implemented $\mathbf{R}$ is 0, the writer w invokes only one operation into $\mathbf{R}$, namely a write of 1, and each reader reads $\mathbf{R}$ at most once (i.e., $\mathbf{R}$ is only a "one-shot" binary register). Moreover, in each of these executions the writer is not malicious (it may only crash) and there is at most one malicious reader; the other $n-1$ readers are correct. Since $\mathcal{I}$ is a linearizable register implementation and the writer of the register is not malicious, these executions of $\mathcal{I}$ must satisfy Properties 1 and 2 of Definition 3.

Let S be the following execution of $\mathcal{I}$ (see Figure 1):

- The writer w is correct.
- All the readers take no steps.
- The writer w invokes a write 1 operation on $\mathbf{R}$. Let s^0 denote the invocation step, and let t_w^0 be the time when s^0 occurs. This step is "local" to w, i.e., it does not invoke any shared register operations.
  During this write operation, w executes a sequence of steps $s^1, ..., s^m$ such that each step s^i is either the reading or the writing of an atomic $[1,n-1]$-register. Let R_i denote the register that w writes or reads in step s^i. Let t_w^i be the time when s^i occurs.
- Since $\mathcal{I}$ is a wait-free implementation and w is correct, w completes its write operation. Let s^{m+1} denote the response step, and let t_w^{m+1} be the time when s^{m+1} occurs. Like s^0, this step is also "local" to w.

▶ **Definition 5.** *For all i, $0 \leq i \leq m+1$, the step s^i of the writer w is* invisible *to a reader x if: (1) s^i is the invocation step s^0, (2) s^i is the response step s^{m+1}, (3) s^i is the reading of an atomic register, or (4) s^i is the writing to an atomic register that is not readable by x.*

Since there are n readers, and the registers that w uses are atomic $[1,n-1]$-registers, every write by w into one of these registers is invisible to one of the readers. So:

▶ **Observation 6.** *For all $0 \leq k \leq m+1$, step s^k is invisible to at least one of the n readers.*

▶ **Definition 7.** *For every k, $0 \leq k \leq m+1$, an execution of $\mathcal{I}$ has property P_k if the following holds:*

1. *The writer w behaves exactly as in S up to and including time t_w^k; then it crashes and takes no steps after time t_w^k. So, w executes steps $s^0, s^1, \ldots, s^k$ and then crashes.*
2. *There is a reader x that is correct and such that:*
   - *Step s^k is invisible to x.*
   - *After time t_w^k, process x starts and completes a read operation on $\boldsymbol{R}$ that returns 1.*
3. *There is a set Z of $n-2$ distinct readers that are correct and take no steps.*

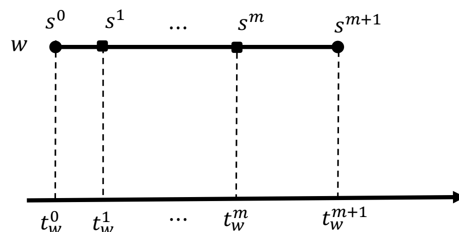


Figure 1 Execution S.

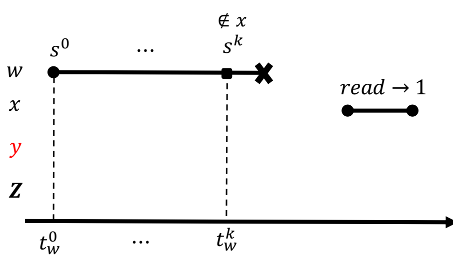


Figure 2 An execution with property P_k.

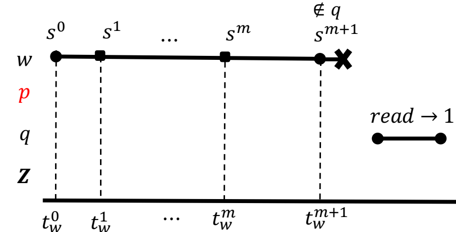


Figure 3 Execution A_{m+1}.

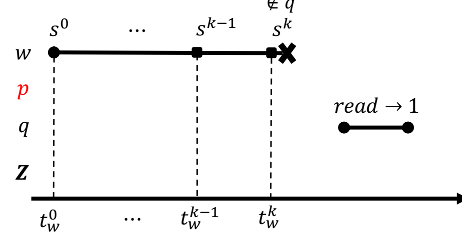


Figure 4 Execution A_k.

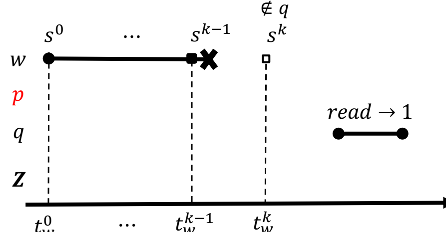


Figure 5 Execution B_{k-1}.

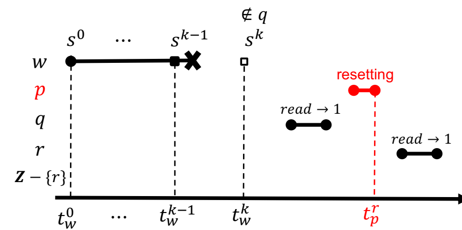


Figure 6 Execution C_{k-1}^r.

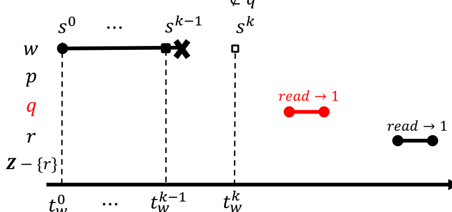


Figure 7 Execution D_{k-1}^r.

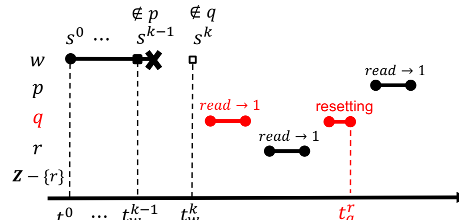


Figure 8 Execution E_{k-1}^r.

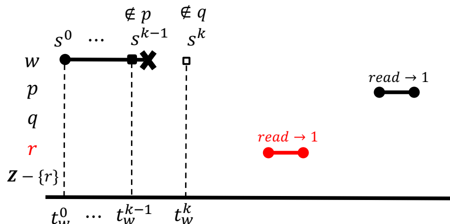


Figure 9 Execution F_{k-1}^r.

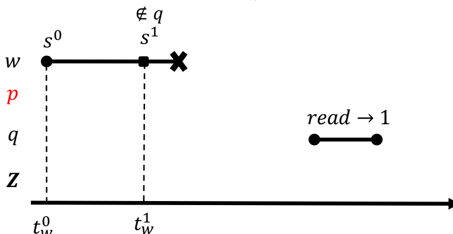


Figure 10 Execution A_1.

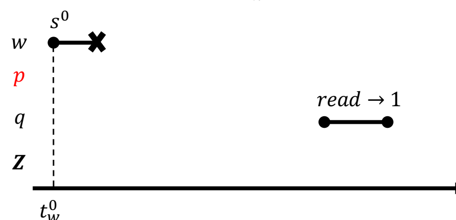


Figure 11 Execution A_0.

Note that since $n \geq 3$, the set Z is not empty. Also note that while the property P_k requires the $n-1$ readers in $\{x\} \cup Z$ to be correct, P_k does not restrict the behavior of the remaining reader; in particular, it may be correct or malicious, and it may or may not take steps.

An execution of $\mathcal{I}$ with property P_k is shown in Figure 2. In this figure and all the subsequent ones, correct readers are in black font, while the reader that may be malicious is colored red; the steps that this process may have taken are not shown in the figure. The "$\notin x$" on top of a step s^i means that s^i is invisible to the reader x. The symbol ✖ indicates where the crash of the writer w occurs.

▷ Claim. For every k, $0 \leq k \leq m+1$, there is an execution of $\mathcal{I}$ that has property P_k.

Proof. We prove the claim by a backward induction on k, starting from $k = m+1$.

Base Case. $k = m+1$. Consider the following execution denoted A_{m+1} (Figure 3):

- The writer w behaves as in execution S up to and including time t_w^{m+1}; then it crashes.
- A reader q is correct. After time t_w^{m+1}, q starts a read operation on $\mathbf{R}$. Since $\mathcal{I}$ is a wait-free implementation, q completes its read operation. Since w is not malicious, and the write operation by w immediately precedes the read operation by q, by the linearizability of $\mathcal{I}$, the read operation by q returns 1.
- There is a set Z of $n-2$ readers that are correct and take no steps, exactly as in S.
- p is the remaining reader.

Since s^{m+1} is a response step, it is invisible to q. So it is clear that A_{m+1} has property P_{m+1}.

Induction Step. Let k be such that $1 \leq k \leq m+1$. Suppose there is an execution A_k of $\mathcal{I}$ that has property P_k (this is the induction hypothesis). We now show that there is an execution A_{k-1} of $\mathcal{I}$ that has property P_{k-1}. We consider two cases, namely, $k > 1$ and $k = 1$.

Case $k > 1$. Since execution A_k of $\mathcal{I}$ satisfies P_k, the following holds in A_k (see Figure 4):

- The writer w behaves as in execution S up to and including time t_w^k; then it crashes.
- There is a reader q that is correct such that step s^k is invisible to q. After time t_w^k, q starts and completes a read operation on $\mathbf{R}$ that returns 1.
- There is a set Z of $n-2$ readers that are correct and take no steps, exactly as in S.
- p is the remaining reader.

Then the following execution B_{k-1} of $\mathcal{I}$ also exists (Figure 5): B_{k-1} is exactly like A_k except that w crashes just before taking step s^k (so B_{k-1} is just A_k with the step s^k "removed").

B_{k-1} is possible because: (1) even though p may have "noticed" the removal of step s^k, p may be malicious (all other readers are correct in this execution), and (2) q cannot distinguish between A_k and B_{k-1} because s^k is invisible to q, and p and all the readers in Z behave as in A_k.

Since $k > 1$, A_k has a step $s^{k-1} \neq s^0$. There are two cases:

Case 1. s^{k-1} *is invisible to* q. Then B_{k-1} is an execution of $\mathcal{I}$ that has the property P_{k-1}, as we wanted to show.

Case 2. s^{k-1} *is visible to* q. Then s^{k-1} is invisible to p or to some reader in Z.

Let r be *any* process in Z. We construct the execution C_{k-1}^r of $\mathcal{I}$ shown in Figure 6: C_{k-1}^r is a continuation of B_{k-1} where, after the correct reader q reads 1, malicious p wipes out any trace of the write steps that it has taken so far, and then a correct process $r \in Z$ reads 1 (this is the only value that r can read, since correct q previously read 1). More precisely:

- C^r_{k-1} is an extension of B_{k-1}.
- After the correct reader q completes its read operation on $\mathbf{R}$, q takes no steps.
- All the readers in $Z - \{r\}$ are correct and take no steps[3].
- After q completes its read operation, p resets all the atomic registers that it can write to their initial values. Process p can do so because it may be malicious (all other readers are correct in this execution). Let t^r_p be the time when p completes all the register resettings.
- A correct reader r starts a read operation on $\mathbf{R}$ after time t^r_p. It takes no steps before this read. Since $\mathcal{I}$ is a wait-free implementation, r completes its read operation. Since w is not malicious and the read operation by correct q returns 1 and precedes the read operation by r, by the linearizability of $\mathcal{I}$, the read operation by r returns 1.

We can now construct the following execution D^r_{k-1} of $\mathcal{I}$ (Figure 7). D^r_{k-1} is obtained from C^r_{k-1} by removing all the steps of p. Despite this removal, q behaves the same as in C^r_{k-1} because q is now malicious. Correct r also behaves as in C^q_{k-1} because it cannot see the removal of p's steps: in both C^r_{k-1} and D^r_{k-1}, r does not "see" any steps of p. More precisely in D^r_{k-1}:

- w behaves exactly as in C^r_{k-1}.
- p is correct and takes no steps. So all its registers retain their initial value.
- All the readers in $Z - \{r\}$ are correct and take no steps as in C^r_{k-1}.
- q behaves the same as in C^r_{k-1}. This is possible because even though q may have "noticed" the removal of p's steps, q may be malicious (all other readers are correct in this execution).
- After possibly malicious q "reads" 1, the correct reader r starts and completes a read operation on $\mathbf{R}$. Since r cannot see the removal of p's steps, and q and all the readers in $Z - \{r\}$ behave the same as in C^r_{k-1}, r cannot distinguish between D^r_{k-1} and C^r_{k-1}. So the read operation by r returns 1 as in C^r_{k-1}.

Note that if s^{k-1} is invisible to process r, then the execution D^r_{k-1} of $\mathcal{I}$ has property P_{k-1}.

Recall that (1) the process r above is an *arbitrary* process in Z, and (2) s^{k-1} is invisible to p or to some reader $r' \in Z$. So there are two cases:

Subcase 2a. *s^{k-1} is invisible to some reader $r' \in Z$.* In the above we proved that the execution $D^{r'}_{k-1}$ of $\mathcal{I}$ has property P_{k-1}, as we wanted to show.

Subcase 2b. *s^{k-1} is invisible to p.* In this case we construct the continuation E^r_{k-1} of D^r_{k-1} shown in Figure 8: after r reads 1, malicious process q wipes out any trace of the write steps that it has taken so far (by reinitializing its registers), and then correct process p applies a read operation to $\mathbf{R}$. By wait freedom, this read operation by p must complete. Since w is not malicious and correct r previously read 1, by linearizability, this read operation by p must return 1.

Finally, we construct the execution F^r_{k-1} of $\mathcal{I}$ by removing all the steps of q from E^r_{k-1} (see Figure 9); so q (which was malicious in E^r_{k-1}) is now a correct process that takes no steps. Despite this removal, r behaves the same as in E^r_{k-1} because r (which was correct in E^r_{k-1}) may now be malicious. Moreover, correct p also behaves as in E^r_{k-1} because it cannot see the removal of q's steps: in both E^r_{k-1} and F^r_{k-1}, p does not "see" any steps of q. So the read operation by p returns 1 as in E^r_{k-1}.

Note that, since s^{k-1} is invisible to p, F^r_{k-1} is an execution of $\mathcal{I}$ that has property P_{k-1}.

[3] If $n = 3$, then the set $Z - \{r\}$ is empty.

Case $k = 1$. By the induction hypothesis, there is an execution A_1 as follows (Figure 10):
- The writer w behaves exactly as in S up to and including time t^1_w; then it crashes.
- After time t^1_w, a correct reader q starts and completes a read operation on $\mathbf{R}$ that returns 1. Furthermore, s^1 is invisible to q.
- There is a set Z of $n - 2$ readers that are correct and take no steps.
- p is the remaining reader.

Then the following execution A_0 of $\mathcal{I}$ also exists (Figure 11): A_0 is like A_1 except that w crashes just before taking step s^1 (so A_0 is just A_1 with the step s^1 "removed"). A_0 is possible because: (1) even though p may have "noticed" the removal of step s^1, p may be malicious (all other readers are correct in this execution), and (2) q cannot distinguish between A_0 and A_1 because s^1 is invisible to q, and p and all the readers in Z behave as in A_1. Since s^0 is an invocation step, it is invisible to q. It is now easy to see that execution A_0 of $\mathcal{I}$ has property P_0, as we wanted to show. ◁

By the claim that we just proved, implementation $\mathcal{I}$ has an execution A_0 with property P_0. By this property, in A_0 process w crashes immediately after the *invocation step* s^0 of its write 1 operation, and some correct reader x later reads the value 1. Since the invocation step s^0 is *invisible* to all the readers (because it does not involve writing any of the shared registers), there is an execution of A'_0 of $\mathcal{I}$ where: (1) w does not take any step at all (so it is not malicious), and (2) a correct reader x reads 1 exactly as in A_0 (because no reader can distinguish between A_0 and A'_0). This execution A'_0 of $\mathcal{I}$ violates the linearizability of $\mathcal{I}$. ◀

It is easy to verify that the above proof holds (without any change) even if all the readers have atomic $[1, n]$-registers that they can write and all processes can read. Thus:

▶ **Theorem 8.** *For all $n \geq 3$, there is* no *wait-free linearizable implementation of a $[1, n]$-register in a system of $n + 1$ processes that are subject to Byzantine failures such that: ss*
- *the writer w of the implemented $[1, n]$-register has atomic $[1, n - 1]$-registers, and every reader has atomic $[1, n]$-registers, and*
- *w can only crash and at most one reader can be malicious.*

5 Register implementation algorithm

We now give an implemention of a $[1, n]$-register from atomic $[1, 1]$-registers in systems with Byzantine process failures; this implementation is linearizable, and it is wait-free provided the writer of the register *or* any number of the readers *but not both* can be faulty. More precisely, it is a *valid* implementation as defined below.

▶ **Definition 9.** *A register implementation is* valid *if it satisfies the following:*
- It is linearizable.
- It is wait-free if the writer is correct or no reader is malicious.

Note that, when executed in a system where processes can only crash, a valid register implementation is linearizable and wait-free (unconditionally).

5.1 Some difficulties to overcome

Note that in a system with Byzantine process failures, implementing a $[1, n]$-register from $[1, 1]$-registers is non-trivial, even if the writer can only crash. To see this, we now illustrate some of the issues that arise. First note that with $[1, 1]$-registers the writer cannot *simultaneously* inform all the readers about a new write. So different readers may have different

views of whether there is a write in progress: some readers may not see it, some readers may see it as still in progress, while other readers may see it as having completed. Thus readers must communicate with each other to avoid "new-old" inversions in the values they read. With non-Byzantine failures, readers can easily coordinate their reads because they can trust the information they pass to each other. With Byzantine failures, however, readers cannot blindly trust what other readers tell them.

For example, suppose a reader q is aware that a write v operation is in progress (say because the writer w directly "told" q about it via the register that they share). To avoid a "new-old" inversion, q checks whether any other reader q' has already read v (because it is possible that from q''s point of view, the write of v already completed). Suppose some q' "warns" q that it has already read the new value v, and so q also reads v. But what if q' is malicious and "lied" to q (and only to q) about having read v? Note that q may be the *only* correct reader currently aware that the write of v is in progress (say because w is slow). Now suppose that a reader q'' that is *not* aware of the write of v also wants to read: if q'' reads the old value of the register this creates a "new-old" inversion with the newer value v that q previously read; but if q'' reads v because q warns q'' that it had read v, then q'' may be reading a value v *that was never written by the correct writer* w: q itself could be malicious and could have "lied" about reading v!

The above is only one of many possible scenarios illustrating why it is not easy to implement a $[1, n]$-register from $[1, 1]$-registers when some readers can be malicious, even if the writer itself is not malicious.

5.2 A recursive solution

To simplify this task, we do not *directly* implement a $[1, n]$-register using *only* $[1, 1]$-registers. Instead, we first give an implementation I_n of a $[1, n]$-register that uses some $[1, n-1]$-registers together with some $[1, 1]$-registers. Then, by replacing the $[1, n-1]$-registers with I_{n-1} implementations, we get an implementation of the $[1, n]$-register that uses some $[1, n-2]$-registers and some $[1, 1]$-registers. By recursing down to $n = 2$, this gives an implementation of the $[1, n]$-register that uses only $[1, 1]$-registers. In other words, we can implement a $[1, n]$-register from $[1, 1]$-registers with a *recursive* construction that gradually reduces the number of readers of the base registers that it uses (all the way down to 1). We now describe this recursive implementation and prove its correctness.

5.3 Implementing a [1,n]-register from [1,n-1]-registers

Algorithm 1 gives an implementation I_n of a $[1, n]$-register that is writable by a process w and readable by every process in $\{p\} \cup Q$, where p is an arbitrary reader and all remaining $n-1$ readers are in Q. We distinguish p from the other readers in Q because p and $q \in Q$ use different procedures for reading the implemented $[1, n]$-register. I_n uses two kinds of registers: *atomic* $[1, 1]$-registers and *implemented* $[1, n-1]$-registers. We will show that I_n is *valid* under the assumption that the $[1, n-1]$-register implementations that it uses are also *valid* (and therefore *linearizable*).

Notation. Recall that if R is an atomic register, all operations applied to R are instantaneous, whereas if R is an implemented register, each operation spans an interval of time, from an invocation to a response. However, since we assume that the $[1, n-1]$-register implementations that I_n uses are valid and therefore *linearizable*, we can think of each operation on an implemented $[1, n-1]$-register as being atomic, i.e., as if it takes effect instantaneously at

Algorithm 1 Implementation I_n of a $[1, n]$-register writable by (an arbitrary) process w and readable by the n processes in $\{p\} \cup Q$, for $n \geq 2$. It uses two $[1, n-1]$-registers and some $[1, 1]$-registers.

```
ATOMIC REGISTERS
   R_wp: [1,1]-register; initially (COMMIT, ⟨0, u_0⟩)
   For every processes q, q' ∈ Q:
        R_qq': [1,1]-register; initially ⟨0, u_0⟩
IMPLEMENTED REGISTERS
   R_wQ: [1,n−1]-register; initially (COMMIT, ⟨0, u_0⟩)
   R_pQ: [1,n−1]-register; initially ⟨0, u_0⟩

LOCAL VARIABLES
   c: variable of w; initially 0
   last_written: variable of w; initially ⟨0, u_0⟩
   previous_k: variable of p; initially 0

    WRITE(u):                 ▷ executed by the writer w
 1:    c ← c + 1
 2:    call W(⟨c, u⟩)
 3:    return done

    READ():   ▷ executed by any reader r in {p} ∪ Q
 4:    call R_r()
 5:    if this call returns some tuple ⟨k, u⟩ then
 6:        return u
 7:    else return ⊥

    W(⟨k, u⟩):                                   ▷ executed by w to do its k-th write
 8:    R_wp ← (PREPARE, last_written, ⟨k, u⟩)
 9:    R_wQ ← (PREPARE, last_written, ⟨k, u⟩)
10:    R_wp ← (COMMIT, ⟨k, u⟩)
11:    R_wQ ← (COMMIT, ⟨k, u⟩)
12:    last_written ← ⟨k, u⟩
13:    return done

    R_p():                                                ▷ executed by reader p
14:    if R_wp = (COMMIT, ⟨k, u⟩) for some ⟨k, u⟩ with k ≥ previous_k then
15:        R_pQ ← ⟨k, u⟩
16:        previous_k ← k
17:        return ⟨k, u⟩
18:    elseif R_wp = (PREPARE, last_written, −) for some last_written then
19:        return last_written
20:    else return ⊥

    R_q():                                                ▷ executed by any reader q ∈ Q
21:    if R_wQ = (COMMIT, ⟨k, u⟩) for some ⟨k, u⟩ then
22:        return ⟨k, u⟩
23:    elseif R_wQ = (PREPARE, last_written, ⟨k, u⟩) for some last_written and some ⟨k, u⟩ then
24:        cobegin
           // THREAD 1
25:            repeat forever
26:                if R_wQ = (COMMIT, ⟨k', −⟩) for some k' ≥ k then
27:                    return ⟨k, u⟩
28:                if R_wQ = (PREPARE, −, ⟨k', −⟩) for some k' > k then
29:                    return ⟨k, u⟩
           // THREAD 2
30:            if R_pQ = ⟨k', −⟩ for some k' ≥ k then
31:                for every process q' ∈ Q do R_qq' ← ⟨k, u⟩
32:                return ⟨k, u⟩
33:            elseif R_q'q = ⟨k', −⟩ for some q' ∈ Q and some k' ≥ k then
34:                if R_pQ = ⟨k', −⟩ for some k' ≥ k then
35:                    for every process q' ∈ Q do R_qq' ← ⟨k, u⟩
36:                return ⟨k, u⟩
37:            else return  last_written
38:        coend
39:    else return ⊥
```

some point during its execution interval [10]. Thus to read or write a register R we use the same notation, irrespective of whether R is atomic or implemented. In particular, in our implementation algorithm (shown in Figure 1) we use the following notation:

- "$R \leftarrow v$" denotes the operation that writes v into R.
- "**if** $R = val$ **then** ..." means "read register R and if the value read is equal to val then ..."

The shared registers used by the implementation are as follows:

- $R_{rr'}$ is an atomic $[1,1]$-register writable by process r and readable by process r'.[4]
- R_{wQ} is an implemented $[1, n-1]$-register writable by w and readable by every $q \in Q$.
- R_{pQ} is an implemented $[1, n-1]$-register writable by p and readable by every $q \in Q$.

Description. The implementation I_n of a $[1, n]$-register from $[1, n-1]$-registers consists of two procedures, namely WRITE() for the writer w, and READ() for each reader r in $\{p\} \cup Q$. To write a value u, the writer w executes WRITE(u). If u is the k-th value written by w, WRITE(u) first forms the unique tuple $\langle k, u \rangle$ and then it calls the lower-level write procedure W($\langle k, u \rangle$) to write this tuple. Intuitively, WRITE() tags the values that it writes with a counter value to make them unique and to indicate in which order they are written.

To read a value, a reader $r \in \{p\} \cup Q$ calls READ(), and this in turn calls a lower-level read procedure $\text{R}_r()$ that reads tuples written by W(). There are two version of the procedure $\text{R}_r()$: one used when $r = p$ and one used when $r \in Q$. If $\text{R}_r()$ returns a tuple of the form $\langle j, v \rangle$, then READ() strips the counter j from the tuple and returns the value v as the value read (otherwise READ() returns $\perp$ to indicate a read failure).

Thus the lower-level procedures W(), $\text{R}_p()$, and $\text{R}_q()$ for each $q \in Q$, are executed to write and read unique tuples of the form $\langle k, u \rangle$. We now describe how these procedures work.

- To execute W($\langle k, u \rangle$), process w first writes (PREPARE, *last_written*, $\langle k, u \rangle$) in the R_{wp} register that p can read, and then in the R_{wQ} register that every process in Q can read; *last_written* is the *last* tuple written by w before $\langle k, u \rangle$ (so *last_written* $= \langle k-1, u' \rangle$ for some u'). Then, w writes (COMMIT, $\langle k, u \rangle$) into R_{wp} and then into R_{wQ}.
- To execute $\text{R}_p()$, process p reads R_{wp} (line 14). If p reads (COMMIT, $\langle k, u \rangle$) with a k at least as big as those it saw before, it returns $\langle k, u \rangle$ as the tuple read (line 17); just before doing so, however, it writes $\langle k, u \rangle$ in the R_{pQ} register that every process $q \in Q$ can read (line 15): intuitively, this is to "warn" them that p read a "new" tuple, to help avoid "new-old" inversion in the tuples read.
  If p reads (PREPARE, *last_written*, $\langle k, u \rangle$) (line 18), then it returns *last_written* as the tuple read (without giving any "warning" about this to processes in Q).
  If p reads anything else from R_{wp}, then it returns $\perp$ (the writer is surely malicious).
- To execute $\text{R}_q()$, process $q \in Q$ reads R_{wQ}. If q reads (COMMIT, $\langle k, u \rangle$) (line 21), it just returns $\langle k, u \rangle$ as the tuple read in line 22 (without "warning" other processes).
  If q reads (PREPARE, *last_written*, $\langle k, u \rangle$) (line 23), then q *cannot* simply return *last_written* as the tuple read: this is because p could have already read (COMMIT, $\langle k, u \rangle$) from R_{wp} and so p could have already read the "newer" tuple $\langle k, u \rangle$ with $\text{R}_p()$. So q must determine whether to return *last_written* or $\langle k, u \rangle$. To do so, q forks two threads and executes them in parallel (we will explain why below).[5]

[4] If $r = r'$, this "shared register" is actually just a local register of process r.
[5] If q does not read values of the form (PREPARE, *last_written*, $\langle k, u \rangle$) or (COMMIT, $\langle k, u \rangle$) from R_{wQ}, then w is surely malicious, and q just returns $\perp$ in line 39.

In Thread 1, process q keeps reading R_{wQ}: if it ever reads (commit, $\langle k', -\rangle$) with $k' \geq k$, or (prepare, $-$, $\langle k', -\rangle$) with $k' > k$, it simply returns $\langle k, u\rangle$ as the tuple read. Note that *if the writer w is correct*, then q *cannot* spin forever in this thread without returning $\langle k, u\rangle$.

In Thread 2, process q first reads the register R_{pQ} to see whether p "warned" processes in Q that it read a tuple at least as "new" as $\langle k, u\rangle$.

- If q sees that R_{pQ} contains a tuple at least as "new" as $\langle k, u\rangle$ (line 30), then q returns $\langle k, u\rangle$ as the tuple read (line 32); but before doing so, q successively writes $\langle k, u\rangle$ in each register $R_{qq'}$ such that $q' \in Q$ (line 31): intuitively, this is to "warn" each process in Q that q read this "new" tuple.
- Otherwise, q reads every $R_{q'q}$ register to avoid a new-old inversion with any tuple read by any process $q' \in Q$: if q sees that some $R_{q'q}$ contains a tuple at least as "new" as $\langle k, u\rangle$ (line 33), then q reads R_{pQ} *again* (line 34) (so q does *not* simply "trust" q' and return $\langle k, u\rangle$!). If q sees that R_{pQ} contains a tuple at least as "new" as $\langle k, u\rangle$ (line 34), then q returns $\langle k, u\rangle$ as the tuple read (line 36); and before doing so q successively writes $\langle k, u\rangle$ to every register $R_{qq'}$ such that $q' \in Q$ (line 35).
- Finally, if q does not see that R_{pQ} or $R_{qq'}$ contain a tuple at least as "new" as $\langle k, u\rangle$ (in lines 30 and 33), then q returns *last_written* (line 37).

Why two parallel threads? In a nutshell, this is to guarantee the wait-freedom of I_n in runs where the writer is correct or no reader is malicious. This is required for our implementation to be valid. It turns out that:

(A) if only Thread 1 is executed, then a faulty writer can block correct readers even if no reader is malicious, and

(B) if only Thread 2 is executed, then malicious readers can block correct readers from returning any value in this thread even if the writer is correct.

But if the writer is correct or no reader is malicious, we can show that every read operation by a correct reader is guaranteed to complete with a return value *in one of the two threads*.

It is easy to see why a faulty writer (even one that just crashes) may block a correct reader in Thread 1. We now explain how malicious readers may impede correct readers in Thread 2.

In Thread 2 readers must read R_{pQ} at least once (in line 30). Recall that (a) R_{pQ} is an *implemented* $[1, n-1]$-register, and (b) we are *only* assuming that this implementation is *valid*. In particular, if the writer p of R_{pQ} crashes *and* some readers of R_{pQ} are malicious, the implementation of R_{pQ} does *not* guarantee the wait-freedom of its read operations. In other words, if p crashes *and* some readers of R_{pQ} are malicious, a correct reader q may block while trying to read R_{pQ}!

Malicious readers may also prevent a correct reader q from reading any tuple in Thread 2 as follows. When q executes $\mathrm{R}_q()$ the following can occur: (1) in line 33, q sees that some $R_{q'q}$ contains $\langle k', -\rangle$ with $k' \geq k$, but (2) in line 34 q sees that R_{pQ} does *not* contain $\langle k', -\rangle$ with $k' \geq k$. We can show that this can occur *only if* at least one of p or q' is malicious. Note that if (1) and (2) indeed occur, then q terminates Thread 2 *without returning any tuple* (because the **if** of line 34 does not have a corresponding **else**).

The correctness of the implementation I_n given by Algorithm 1 is stated in Theorem 10. The proof of this theorem is given in [11].

▶ **Theorem 10.** *For all $n \geq 2$, I_n is an implementation of a $[1, n]$-register from implemented $[1, n-1]$-registers and atomic $[1, 1]$-registers. I_n is valid if the implementations of the $[1, n-1]$-registers that it uses (namely, R_{wQ} and R_{pQ}) are valid.*

It's worth noting that for the case $n = 2$, there is a simple implementation $I_2{}'$ that is stronger than the I_2 implementation given by Algorithm 1: in contrast to I_2, $I_2{}'$ is *unconditionally* wait-free. The implementation $I_2{}'$ is given by Algorithm 3 in Appendix A. Note that Algorithm 3 is a simple version of Algorithm 1: the set of readers Q now contains only one process q, and so preventing new-old inversions is much easier.

▶ **Theorem 11.** *The implementation $I_2{}'$ (given by Algorithm 3 in Appendix A) is a wait-free linearizable implementation of a $[1, 2]$-register from atomic $[1, 1]$-registers.*

5.4 Implementing a [1,n]-register from atomic [1,1]-registers

We now show our main "possibility" result: in a system with Byzantine process failures, there is an implementation of a $[1, n]$-register from atomic $[1, 1]$-registers that is linearizable and wait-free provided that the writer *or* any number of readers, but *not both*, can fail. In fact we show the following stronger result:

▶ **Theorem 12.** *For all $n \geq 2$, in a system of $n + 1$ processes that are subject to Byzantine failures, there is a* valid *implementation $\mathcal{I}_n$ of a $[1, n]$-register from atomic $[1, 1]$-registers.*

Proof. We show Theorem 12 by induction on n.

Base Case. Let $n = 2$. Consider the implementation I_2 of Theorem 10. Since $n = 2$, the set Q now contains only one process. So each register R_{wQ} and R_{pQ} in I_2 can be implemented directly by an *atomic* $[1, 1]$-register. Since these are *valid* implementations of R_{wQ} and R_{pQ}, there is a valid implementation $\mathcal{I}_2$ of a $[1, 2]$-register from atomic $[1, 1]$-registers.[6]

Induction Step. Let $n > 2$. Suppose there is a valid implementation $\mathcal{I}_{n-1}$ of a $[1, n - 1]$-register that uses only atomic $[1, 1]$-registers. We must show there is a valid implementation $\mathcal{I}_n$ of a $[1, n]$-register that uses only atomic $[1, 1]$-registers.

By Theorem 10, there is an implementation I_n of a $[1, n]$-register that uses:

1. two implemented $[1, n - 1]$-registers (namely, of registers R_{wQ} and R_{pQ}), and
2. some atomic $[1, 1]$-registers

such that I_n is valid if the implementations of the $[1, n - 1]$-registers R_{wQ} and R_{pQ} are valid. Implement R_{wQ} and R_{pQ} in I_n using the valid implementation $\mathcal{I}_{n-1}$ ($\mathcal{I}_{n-1}$ exists by our induction hypothesis). This gives an implementation $\mathcal{I}_n$ of a $[1, n]$-register that uses only atomic $[1, 1]$-registers (because $\mathcal{I}_{n-1}$ uses only atomic $[1, 1]$-registers). Since the implementations of R_{wQ} and R_{pQ} are valid, $\mathcal{I}_n$ is valid. ◀

Since $\mathcal{I}_n$ is valid, it is linearizable (no matter which processes fail and how); and it is wait-free *provided the writer is correct or no reader is malicious.* This matches the impossibility result given by Theorem 4 in Section 4.

[6] To show that $\mathcal{I}_2$ exists, we could also use the wait-free and linearizable implementation $I_2{}'$ mentioned in Theorem 11.

Algorithm 2 Implementation $\mathcal{I}_s$ of a $[1, n]$-register writable by process w and readable by a set P of n processes in a system with unforgeable signatures. $\mathcal{I}_s$ uses atomic $[1, 1]$-registers.

ATOMIC REGISTERS
For every processes i and j:
R_{ij}: atomic $[1, 1]$-register; initially $\langle 0, u_0 \rangle_w$.

LOCAL VARIABLES
c: variable of w; initially 0
tuples: variable of each $p \in P$; initially $\emptyset$.

```
    WRITE(u):                          ▷ executed by the writer w
1:     c ← c + 1
2:     call W(⟨c, u⟩_w)
3:     return done

    READ():                            ▷ executed by any reader p in P
4:     call R()
5:     if this call returns some tuple ⟨k, u⟩_w then
6:         return u
7:     else return ⊥

    W(⟨k, u⟩_w):                       ▷ executed by w to do its k-th write
8:     for every process i ∈ P do
9:         R_wi ← ⟨k, u⟩_w                 ▷ ⟨k, u⟩ signed by w
10:    return done

    R():                               ▷ executed by any reader p ∈ P
11:    tuples ← ∅
12:    for every process i ∈ {w} ∪ P do
13:        if R_ip = ⟨ℓ, val⟩_w for some ⟨ℓ, val⟩ validly signed by w then
14:            tuples ← tuples ∪ {⟨ℓ, val⟩_w}
15:    ⟨k, u⟩_w ← tuple ⟨ℓ, val⟩_w with maximum sequence number ℓ in tuples
16:    for every process i ∈ P do
17:        R_pi ← ⟨k, u⟩_w
18:    return ⟨k, u⟩_w
```

6 Implementation for systems with digital signatures

Algorithm 2 gives a linearizable and wait-free implementation $\mathcal{I}_s$ of a $[1, n]$-register that is writable by process w and readable by a set P of n processes. $\mathcal{I}_s$ uses unforgeable signatures of processes (actually only w does) and *atomic* $[1, 1]$-registers between each pair of processes.

As in Algorithm 1, to write a value u the writer w first adds a counter k to form a tuple $\langle k, u \rangle$. It then signs $\langle k, u \rangle$, and the signed tuple is denoted $\langle k, u \rangle_w$. As before, the actual write and read operations are done by lower-level procedures W() and R(), which work as follows:

- To execute W($\langle k, u \rangle_w$), the writer w simply writes $\langle k, u \rangle_w$ in R_{wi} for every process i.
- To execute R(), the process p first reads the $[1, 1]$-register R_{ip} of every process i to collect a set *tuples* of the tuples with valid signature of w. Then p selects the tuple $\langle k, u \rangle_w$ with maximum sequence number k in *tuples*, and return this tuple; but before doing so p writes $\langle k, u \rangle_w$ into every $[1, 1]$-register R_{pi} to notify every process i that it read this tuple.

The correctness of the implementation $\mathcal{I}_s$ given by Algorithm 2 is stated in Theorem 13. The proof of this theorem is given in [11].

▶ **Theorem 13.** *Consider a system where processes are subject to Byzantine failures and can use unforgeable signatures. For every $n \geq 2$, $\mathcal{I}_s$ is a wait-free linearizable implementation of a $[1, n]$-register from atomic $[1, 1]$-registers that tolerates any number of faulty processes.*

7 Concluding remarks

The implementation of registers from weaker registers is a basic problem in distributed computing that has been extensively studied in the context of processes with crash failures. In this paper, we investigated this problem in the context of Byzantine processes failures, with and without process signatures. We first proved that there is no wait-free linearizable implementation of a $[1, n]$-register from atomic $[1, n-1]$-registers. In fact, we showed that this impossibility holds even if every process except the writer can use atomic $[1, n]$-registers, and even under the assumption that the writer can only crash and *at most one* reader can be malicious. This is in sharp contrast to the situation in systems with crash failures only, where there *is* a wait-free linearizable implementation of a $[1, n]$-register even from *safe* $[1, 1]$-registers [14].

In light of this strong impossibility result, we gave an implementation of a $[1, n]$-register from atomic $[1, 1]$-registers that is linearizable (intuitively, "safe") under any combination of Byzantine process failures, but is wait-free (intuitively, "live") only under the assumption that the writer is correct or no reader is malicious; this matches the impossibility result. We also gave an implementation that uses process signatures, and is wait-free and linearizable under any number and combination of Byzantine process failures.

Perhaps surprisingly, none of the above results refers to a ratio of faulty vs. correct processes, such as $n/3$ or $n/2$, that we typically encounter in results that involve Byzantine processes. For example, Mostéfaoui *et al.* [15] prove that one can implement an f-resilient $[1, n]$-register in *message-passing* systems with Byzantine process failures if and only if $f < n/3$. As an other example, Cohen and Keidar [6] show that if $f < n/2$, one can use atomic $[1, n]$-registers to get f-resilient implementations of reliable broadcast, atomic snapshot, and asset transfer objects in systems with Byzantine process failures.

It is worth noting that, since atomic $[1, 1]$-registers can simulate message-passing channels, one can use the f-resilient implementation of a $[1, n]$-register for *message-passing* systems given in [15], to obtain an *f-resilient* implementation of a $[1, n]$-register using atomic $[1, 1]$-registers. But f-resilient implementations (such as the ones given in [6, 15]) require *every* correct process to help the execution of *every* operation, even the operations of *other* processes. In contrast, with wait-free object implementations in shared-memory systems, processes that do not have ongoing operations take no steps.

References

1 Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The impact of RDMA on agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, PODC '19, pages 409–418, 2019. doi:10.1145/3293611.3331601.

2 James H Anderson and Mohamed G Gouda. *The virtue of patience: Concurrent programming with and without waiting.* Citeseer, 1990.

3 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995. doi:10.1145/200836.200869.

4 B. Bloom. Constructing two-writer atomic registers. *IEEE Trans. Comput.*, 37(12):1506–1514, December 1988.

5 James E. Burns and Gary L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 222–231, 1987. doi:10.1109/12.9729.

6 Shir Cohen and Idit Keidar. Tame the Wild with Byzantine Linearizability: Reliable Broadcast, Snapshots, and Asset Transfer. In *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209, pages 18:1–18:18, 2021. doi:10.4230/LIPIcs.DISC.2021.18.

7 Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, 1978. doi:10.1007/3-540-08755-9_9.

8 S. Haldar and K. Vidyasankar. Constructing 1-writer multireader multivalued atomic variables from regular variables. *J. ACM*, 42(1):186–203, January 1995. doi:10.1145/200836.200871.

9 Maurice P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 276–290, 1988. doi:10.1145/62546.62593.

10 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

11 Xing Hu and Sam Toueg. On implementing swmr registers from swsr registers in systems with byzantine failures, 2022. doi:10.48550/arXiv.2207.01470.

12 Amos Israeli and Amnon Shaham. Optimal multi-writer multi-reader atomic register. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '92, pages 71–82, 1992. doi:10.1145/135419.135435.

13 Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *J. ACM*, 45(3):451–500, May 1998. doi:10.1145/278298.278305.

14 Leslie Lamport. On interprocess communication Parts I–II. *Distributed Computing*, 1(2):77–101, 1986. doi:10.1007/BF01786227.

15 Achour Mostéfaoui, Matoula Petrolia, Michel Raynal, and Claude Jard. Atomic read/write memory in signature-free byzantine asynchronous message-passing systems. *Theory of Computing Systems*, 60, May 2017. doi:10.1007/s00224-016-9699-8.

16 Richard Newman-Wolfe. A protocol for wait-free, atomic, multi-reader shared variables. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 232–248, 1987. doi:10.1145/41840.41860.

17 Gary L. Peterson. Concurrent reading while writing. *ACM Trans. Program. Lang. Syst.*, 5(1):46–55, January 1983. doi:10.1109/SFCS.1986.11.

18 Gary L. Peterson and James E. Burns. Concurrent reading while writing ii: The multi-writer case. In *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*, SFCS '87, pages 383–392, 1987. doi:10.1109/SFCS.1987.15.

19 Ambuj K. Singh, James H. Anderson, and Mohamed G. Gouda. The elusive atomic register revisited. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 206–221, 1987. doi:10.1145/41840.41858.

20 K. Vidyasankar. Converting Lamport's regular register to atomic register. *Inf. Process. Lett.*, 28(6):287–290, August 1988. doi:10.1016/0020-0190(88)90175-5.

21 K. Vidyasankar. A very simple construction of 1-writer multireader multivalued atomic variable. *Inf. Process. Lett.*, 37(6):323–326, March 1991. doi:10.1016/0020-0190(91)90149-C.

22 Paul M. B. Vitanyi and Baruch Awerbuch. Atomic shared register access by asynchronous hardware. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 233–243, 1986.

A A wait-free linearizable implementation of a [1,2]-register from atomic [1,1]-registers.

Algorithm 3 gives a *wait-free* linearizable implementation ${I_2}'$ of a $[1,2]$-register from atomic $[1,1]$-registers. This algorithm is a simpler version of Algorithm 1 for the valid implementation I_n of a $[1,n]$-register: ${I_2}'$ has only two readers, namely p and q, so preventing new-old inversions among readers is easier. In contrast to Algorithm 1, the code of Algorithm 3 has no parallel threads.

Since the code of Algorithm 3 does not contain any loop or wait statement, it is clear that every call to the WRITE() and READ() procedures by any correct process terminates with a return value in a bounded number of steps; i.e., the implementation ${I_2}'$ is wait-free. The proof that it is a *linearizable* implementation is given in [11]. So we have:

▶ **Theorem 11.** *The implementation ${I_2}'$ (given by Algorithm 3 in Appendix A) is a wait-free linearizable implementation of a* $[1,2]$*-register from atomic* $[1,1]$*-registers.*

Algorithm 3 Implementation ${I_2}'$ of a $[1,2]$-register writable by w and readable by p and q. ${I_2}'$ uses some $[1,1]$-registers.

```
ATOMIC REGISTERS
  R_wp: [1,1]-register; initially (COMMIT, ⟨0, u_0⟩)
  R_wq: [1,1]-register; initially (COMMIT, ⟨0, u_0⟩)
  R_pq: [1,1]-register; initially ⟨0, u_0⟩

LOCAL VARIABLES
  c: variable of w; initially 0
  last_written: variable of w; initially ⟨0, u_0⟩
  last_read: variable of q initially ⟨0, u_0⟩

    WRITE(u):                 ▷ executed by the writer w
1:    c ← c + 1 s
2:    call W(⟨c, u⟩)
3:    return done

    READ():                   ▷ executed by any reader r ∈ {p, q}
4:    call R_r()
5:    if this call returns some tuple ⟨k, u⟩ then
6:        return u
7:    else return ⊥

    W(⟨k, u⟩):                                ▷ executed by w to do its k-th write
8:    R_wp ← (PREPARE, last_written, ⟨k, u⟩)
9:    R_wq ← (PREPARE, last_written, ⟨k, u⟩)
10:   R_wp ← (COMMIT, ⟨k, u⟩)
11:   R_wq ← (COMMIT, ⟨k, u⟩)
12:   last_written ← ⟨k, u⟩
13:   return done

    R_p():                                    ▷ executed by reader p
14:   if R_wp = (COMMIT, ⟨k, u⟩) for some ⟨k, u⟩ then
15:       R_pq ← ⟨k, u⟩
16:       return ⟨k, u⟩
17:   elseif R_wp = (PREPARE, last_written, −) for some last_written then
18:       return last_written
19:   else return ⊥

    R_q():                                    ▷ executed by reader q
20:   if R_wq = (COMMIT, ⟨k, u⟩) for some ⟨k, u⟩ then
21:       return ⟨k, u⟩
22:   elseif R_wq = (PREPARE, last_written, ⟨k, u⟩) for some last_written and some ⟨k, u⟩ then
23:       if R_pq = ⟨k′, −⟩ for some k′ ≥ k then
24:           last_read ← ⟨k, u⟩
25:           return ⟨k, u⟩
26:       elseif last_read = ⟨k′, −⟩ and some k′ ≥ k then
27:           return ⟨k, u⟩
28:       else
29:           return last_written
30:   else return ⊥
```

Space-Stretch Tradeoff in Routing Revisited

Anatoliy Zinovyev
Boston University, MA, USA

Abstract

We present several new proofs of lower bounds for the space-stretch tradeoff in labeled network routing.

First, we give a new proof of an important result of Cyril Gavoille and Marc Gengler that any routing scheme with stretch < 3 must use $\Omega(n)$ bits of space at some node on some network with n vertices, even if port numbers can be changed. Compared to the original proof, our proof is significantly shorter and, we believe, conceptually and technically simpler. A small extension of the proof can show that, in fact, any constant fraction of the n nodes must use $\Omega(n)$ bits of space on some graph.

Our main contribution is a new result that if port numbers are chosen adversarially, then stretch $< 2k+1$ implies some node must use $\Omega\left(n^{\frac{1}{k}} \log n\right)$ bits of space on some graph, assuming a girth conjecture by Erdős.

We conclude by showing that all known methods of proving a space lower bound in the labeled setting, in fact, require the girth conjecture.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Data structures design and analysis; Mathematics of computing → Discrete mathematics

Keywords and phrases Compact routing, labeled network routing, lower bounds

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.37

1 Introduction

Network routing is an important subject in the field of distributed algorithms. Given a network of nodes (routers) connected arbitrarily, the goal is to be able to transmit information between any two nodes in the network. A network data packet must usually traverse multiple routers in order to reach its destination, and these intermediate routers must be able to make a decision as to where to forward the packet next based only on the packet and some local information. This local information is usually referred to as a routing table or, more precisely, a routing program, and one of the goals is to bound its size. This is particularly important in practice since the memory inside a router must be fast and is therefore expensive. Another goal one might pursue is the quality of the routes chosen by the routers. The route quality is commonly characterized by the stretch factor defined as the ratio between the length of the route and the shortest distance between the two nodes. In practice, it corresponds to latency in a network that does not experience congestion.

In this paper we consider the problem of labeled routing: what is the best possible tradeoff between routing program sizes and the stretch factor given that we are allowed to assign arbitrary (but not long) labels to nodes which the sender must include in the routing header? Although this model is of little practical interest since one doesn't want to reconfigure the labels each time the network changes, labeled routing schemes often serve as an ingredient in name-independent routing: first a label is retrieved from a distributed dictionary after which a labeled routing scheme is used [12, 5, 4, 1]. Additionally, the lower bounds for the labeled routing setting imply the same lower bounds for the name-independent setting, where node labels are chosen adversarially.

© Anatoliy Zinovyev;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 37; pp. 37:1–37:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1.1 Some general notation

For a natural number n, define $[n] = \{1, 2, ..., n\}$. For any vertex v, denote $\mathcal{N}(v)$ to be the set of neighbors of v. Define $\deg(v) = |\mathcal{N}(v)|$, the degree of v.

1.2 Model

We deal with undirected connected graphs G with n nodes (routers). Let $G = (V, E)$ where $V = [n]$. Each $x \in V$ possesses a unique label (name) $l_x \in \{0, 1\}^*$, as well as a routing program R_x. The goal of a (distributed) routing algorithm is to route data from any node x to any node y given only the destination's label l_y. Routers are allowed to attach an arbitrary header to messages and modify the header at each hop. Each router x has network ports numbered $\{1, 2, ..., \deg(x)\}$ to which edges are attached. This port-edge relation is a bijection: each edge is assigned exactly one port and each port is assigned exactly one edge. For $x \in V$ and $1 \leq t \leq \deg(x)$ define $\tau_x(t) = y$ where y is the node such that $\{x, y\}$ is the edge assigned to port t at node x. So, $\tau_x : [\deg(x)] \to \mathcal{N}(x)$ is a bijection. The routing program R_x at node x accepts the header of the incoming message and the incoming port number as input and outputs the new header and the outgoing port number: if $R_x(h, q) = (h', p)$, a message with header h that comes in through port q leaves the router with header h' through port p.

Formally, a routing scheme for G with node labels $\mathcal{L} = (l_1, l_2, ..., l_n)$ and port functions $\mathcal{T} = (\tau_1, \tau_2, ..., \tau_n)$ is a collection of programs $\mathcal{R} = (R_1, R_2, ..., R_n)$ such that for all $x, y \in V$ there exists a finite walk $x = v_0, v_1, ..., v_t = y$ in G, a sequence $h_0, h_1, ..., h_t$ of headers, and two sequences $q_0, ..., q_t$ and $p_1, ..., p_{t+1}$ of port numbers such that:

- for all $0 \leq i < t$:
  - $R_{v_i}(h_i, q_i) = (h_{i+1}, p_{i+1})$
  - $\tau_{v_i}(p_{i+1}) = v_{i+1}$
  - $\tau_{v_{i+1}}(q_{i+1}) = v_i$
- $h_0 = l_y$
- $q_0 = 0$
- $R_{v_t}(h_t, q_t) = (\epsilon, 0)$

This model can take different variations depending on whether the node labels and the port assignment are adversarially chosen or not. In the most adversarial setting, both assignments are given as part of the input and the goal is to construct a correct routing scheme. In the least adversarial setting, only the graph G is given, and the goal is to generate a routing scheme along with node labels and a port assignment. Models where only one assignment is adversarial can also be considered. Generally, minimizing the size of routing programs, node labels and headers is desirable. In this paper, we work with two models: one has non-adversarial port assignment, and the other has adversarial port assignment. Both, however, are models for labeled routing; i.e., node labels are non-adversarial.

Finally, we formally define routing stretch. We say a routing scheme has stretch s if for all x, y, $d_R(x, y) \leq s \cdot d_G(x, y)$ where $d_R(x, y)$ is the length of the path from x to y taken by the routing scheme, and $d_G(x, y)$ is the distance between x and y in G.

1.3 Known results

Many routing schemes have been proposed in the literature. The state of the art for labeled routing includes [17] by Thorup and Zwick who for every integer $k \geq 1$ describe a scheme that uses $\tilde{O}(n^{\frac{1}{k}})$ bits of space at every node and has stretch $4k - 5$, [8] by Chechik who shows a scheme using $\tilde{O}(n^{\frac{1}{k}} \log D)$ bits of space at every node and having stretch $c \cdot k$ for

some $c < 4$ for sufficiently large integer k, and [15] by Roditty and Tov who show a routing scheme using $\tilde{O}\left(\frac{1}{\epsilon} n^{\frac{1}{k}} \log D\right)$ bits of space at each node and having stretch $4k - 7 + \epsilon$ for every integer $k \geq 1$; D is defined to be the normalized network diameter.

Surprisingly, there are also name-independent routing schemes with similar characteristics. An optimal stretch-3 name-independent routing scheme that stores $\tilde{O}\left(\sqrt{n}\right)$ bits at each node is known [4]. Additionally, [3] constructs a routing scheme with $O(k)$ stretch that uses $\tilde{O}\left(n^{\frac{1}{k}}\right)$ bits of space at each node.

Lower bounds for routing schemes also exist and are the focus of this paper. The first such lower bound appeared in the work of Peleg and Upfal [14] who showed that for any $s \geq 1$, any stretch-s routing scheme with adversarial port assignment on n nodes must use a total of $\Omega\left(n^{1+\frac{1}{s+2}}\right)$ bits, and thus $\Omega\left(n^{\frac{1}{s+2}}\right)$ bits at some node. Gavoille and Perennes [11] prove that $\Omega\left(n^2 \log n\right)$ total bits is needed for shortest path routing, even when port numbers can be chosen by the designer; however, node labels are assumed to be $[n]$. Buhrman et al. [7] explore upper and lower bounds under various routing models. In the standard model, that we consider here, they prove that shortest path routing requires $\Omega\left(n^2\right)$ bits of total state, even when ports can be chosen by the designer, and labels can be arbitrary (short) bit strings. Gavoille and Gengler [10] achieve the same lower bound under the same model, but for any stretch $s < 3$. Finally, Thorup and Zwick [17] claim a lower bound of $\Omega\left(n^{1+\frac{1}{k}}\right)$ total bits of space for stretch $s < 2k + 1$ when the port assignment is adversarial, but their proof idea does not seem to work in the standard model. We discuss their claim and proof idea in the next subsection, and present a proof which works under the standard model, with a $\log n$ factor improvement. Table 1 demonstrates known results for labeled routing. Note that a lower bound that works for the non-adversarial port assignment implies the same lower bound for the adversarial port assignment.

Table 1 Known lower bounds for labeled routing.

(a) Non-adversarial (+ adversarial) ports.

Work	*Stretch*	*Total memory (bits)*	*Local memory (bits)*	*Notes*
Gavoille and Perennes [11]	$< 5/3$	$\Omega\left(n^2 \log n\right)$	$\Omega(n \log n)$ on $\Omega(n)$ nodes	Node labels are $[n]$
Buhrman et al. [7]	1	$\Omega\left(n^2\right)$	$\Omega(n)$ on $\Omega(n)$ nodes	
Gavoille and Gengler [10]	< 3	$\Omega\left(n^2\right)$	$\Omega(n)$	
This paper	< 3	$\Omega\left(n^2\right)$	$\Omega(n)$ on cn nodes, $\forall 0 < c < 1$	

(b) Adversarial ports.

Work	*Stretch*	*Total memory (bits)*	*Local memory (bits)*	*Notes*
Peleg and Upfal [14]	$s \geq 1$	$\Omega\left(n^{1+\frac{1}{s+2}}\right)$	$\Omega\left(n^{\frac{1}{s+2}}\right)$	
Thorup and Zwick [17]	$< 2k + 1$	$\Omega\left(n^{1+\frac{1}{k}}\right)$	$\Omega\left(n^{\frac{1}{k}}\right)$	Works in a different model
This paper	$< 2k + 1$	$\Omega\left(n^{1+\frac{1}{k}} \log n\right)$	$\Omega\left(n^{\frac{1}{k}} \log n\right)$	

All known lower bound proofs for stretch $s > 1$ (Peleg and Upfal [14], Thorup and Zwick [17], our proof in section 3) rely on dense graphs of large girth. In an unweighted graph where all cycles have length $> s + 1$, routing with stretch s between neighbor vertices x and y must involve the edge $\{x, y\}$. Thus, x must "know" its neighbor y, and this is what proofs rely on. Currently, however, the existence of dense graphs of large girth is an open question. It is worth noting that Abraham et al. [2] overcome this conjecture in the name-independent model and unconditionally show a $\Omega\left((n \log n)^{\frac{1}{k}}\right)$-bit local memory requirement if weighted networks are allowed.

1.4 Our contribution

1.4.1 Lower bounds with non-adversarial port numbers

Our first contribution is a simpler proof of the result of Gavoille and Gengler ([10], with the full proof in their research report [9]) that shows that any routing scheme with stretch < 3 must use a total of $\Omega(n^2)$ bits of routing state, and thus $\Omega(n)$ bits at some node, on some network with n nodes, even if nodes and port numbers are allowed to be renamed. Our proof, presented in section 2, is three times shorter and, we believe, significantly simpler. This is due to a simpler class of graphs considered, easier reasoning and calculations.

With a bit of additional effort, we are also able to show that, in fact, any constant fraction of the n nodes must use $\Omega(n)$ bits of space on some graph. Previously, similar results were obtained in [11] and [7] for shortest path routing.

[11] proves a total memory requirement of $\Omega(n^2 \log n)$ bits with local memory requirement of $\Omega(n \log n)$ bits on $\Omega(n)$ nodes. We note that $\Omega(n^2 \log n)$-bit total memory requirement implies $\Omega(n \log n)$ bits on $\Omega(n)$ nodes using the following informal generic argument. Suppose the total memory requirement is $c_0 n^2 \log n$ bits, the shortest path routing table can be described in $c_1 n \log n$ bits at each node, and there are x nodes that store $> c_2 n \log n$ bits. Then $x \cdot c_1 n \log n + (n-x) \cdot c_2 n \log n \geq c_0 n^2 \log n$ which implies $x \geq \frac{c_0 - c_2}{c_1 - c_2} n$. Setting $c_2 = \frac{c_0}{2}$ achieves the result. The same reasoning can be used to show that the $\Omega(n^2)$ total memory proved implies $\Omega(n)$ bits at $\Omega\left(\frac{n}{\log n}\right)$ nodes, but this is the best possible generic argument. Indeed, you could have $O\left(\frac{n}{\log n}\right)$ nodes storing $O(n \log n)$ bits and all other nodes storing 0 bits. To show that more nodes must store $\Omega(n)$ bits, one needs a deeper argument.

[7] proves a local memory requirement of $\Omega(n)$ bits on $\Omega(n)$ nodes which can easily be extended to any constant fraction of all nodes. It uses a Kolmogorov random graph and assuming too many nodes have small routing programs reaches a contradiction.

It also seems that Gavoille and Gengler's proof [10] can be extended to show a local memory requirement of $\Omega(n)$ bits on any constant fraction of all nodes. We provide an extension in the context of our proof for completeness.

1.4.2 Lower bounds with adversarial port numbers

Our second contribution is a proof of a claim by Thorup and Zwick that if ports are assigned adversarially, for any integer $k \geq 1$, any labeled routing scheme with stretch $< 2k+1$ requires a total of $\Omega\left(n^{1+\frac{1}{k}}\right)$ bits of space, and so $\Omega\left(n^{\frac{1}{k}}\right)$ bits of space at some node, in the worst case, assuming a well-known conjecture by Erdős regarding the existence of dense graphs with large girth. This claim deserves special discussion.

In [17], Thorup and Zwick make this claim, deferring the full proof to the full version of the paper which never appeared. It was stated, however, that the result should follow easily from their other paper [16] about compact distance oracles. A compact distance oracle is a small data structure for a graph that for any vertices u and v is able to return an approximate distance (up to some constant factor) between them. Their lower bound for distance oracles is now a standard incompressibility argument, perhaps first introduced by Matoušek [13], and it goes as follows. Take a graph with m edges and girth $2k+2$ and consider all 2^m subgraphs. For any two subgraphs, there must be an edge $\{u, v\}$ in one that is absent in the other. If the distance oracle satisfies stretch $< 2k+1$, then it must return distance $[1, 2k+1)$ for u, v in the first graph, and distance $\geq 2k+1$ in the second graph. Thus, each subgraph must have a distinct distance oracle data structure, one of them of size m bits. So, it appears that the lower bound proof for routing that Thorup and Zwick had in mind is assuming the existence of a routing scheme with stretch $< 2k+1$ of total size $< m$ bits, and constructing

distance oracle data structures: given a graph with a compact routing scheme, construct a program that for any vertices u, v traces the route from u to v returned by the routing scheme and reports the length of the route. Clearly, if a routing scheme can be encoded with $< m$ bits, then $< 2^m$ routing schemes are needed to satisfy all graphs, and $< 2^m$ distance oracle programs are needed, a contradiction. The problem with this argument is that one cannot trace the route from u to v given the routing scheme alone. Each routing program returns the port toward the next node, but we do not know what the next node is, and encoding the port assignment takes at least m bits which would break the proof.

Obviously, however, this argument works in the model where the port must equal to the node label it leads to, or if the ports are $\log n$ bit strings chosen adversarially, but this is less natural than providing the router with a port in $\{1, ..., |\mathcal{N}(v)|\}$, and is perhaps more suited for wireless networks. Given that this lower bound was mentioned in the context of the standard model in so many works ([12, 2, 5, 4, 1, 8]), we set out to close this gap and prove the result in the standard, more natural, model. Instead of considering many graphs of large girth, we fix one such dense graph, consider all possible port assignments, and show that a single routing scheme cannot satisfy many of them; thus many routing schemes are needed. Our bound is also slightly stronger: assuming Erdős' conjecture, stretch $< 2k + 1$ requires a total of $\Omega\left(n^{1+\frac{1}{k}} \log n\right)$ bits of space. This implies that the stretch-3 routing scheme of Thorup and Zwick [17] has optimal up to $\sqrt{\log n}$ factor per-node space requirement.

To the best of our knowledge, the best previous lower bound for stretch $s \geq 3$ routing with adversarial ports in the standard model is given by Peleg and Upfal who showed a total memory requirement of $\Omega\left(n^{1+\frac{1}{s+2}}\right)$ [14]. Their proof is based on a probabilistic algorithm for constructing dense graphs with large girth. In contrast, our proof decouples the existence of such graphs and the lower bound argument, which lets us use the best known results regarding the existence of such graphs. In particular, the graph construction in [14] can be used in conjunction with the proof in this paper to obtain the lower bound result in [14], but better graph constructions are available.

1.4.3 Known techiques require girth conjecture

Finally, in section 4 we show that all known methods for proving a space lower bound, that is, finding the number of distinct routing schemes necessary, actually require the existence of dense graphs with large girth.

2 Lower bounds with non-adversarial port numbers

We are interested in the complexity of a routing scheme defined by $\max\left\{|R_v| : v \in V\right\}$ – the size of the largest routing algorithm.

The idea of the lower bound, as in [10], is to consider a family of graphs of girth 4 (where all cycles have length ≥ 4) and show that $2^{\Omega(n^2)}$ configurations of routing programs $\mathcal{R}$ are needed to satisfy all graphs in the family with stretch < 3. From this it follows that at least one routing program must have size $\Omega(n)$ bits, as shown by the following lemma.

▶ **Lemma 1.** *Let n be a positive integer, and let $S \subseteq \left(\{0, 1\}^*\right)^n$ be a collection of n-tuples of binary strings with $|S| > 0$. Then some string must have length $\geq \frac{\log|S|}{n} - 1$.*

Proof. Suppose all strings have length $< \frac{\log|S|}{n} - 1$. Then all strings have length $\leq L = \left\lfloor \frac{\log|S|}{n} - 1 \right\rfloor$. But then $|S| \leq \left(2^{L+1} - 1\right)^n < \left(2^{L+1}\right)^n \leq \left(2^{\frac{\log|S|}{n}}\right)^n = |S|$ which is a contradiction. ◀

▶ **Theorem 2.** *There exists a function $f(n) \in \Omega(n)$ such that if node labels have size $\leq f(n)$ bits, then there exists an unweighted graph with n nodes for which any routing scheme with stretch < 3 contains a routing program of size $\Omega(n)$ bits.*

Proof. Let $n = 2q + 2$ (the case where n is odd is analogous), $V = A \cup B$ where $A = \{a_0, a_1, ..., a_q\}$ and $B = \{b_0, b_1, ..., b_q\}$. Define $\mathcal{G}$ to be the set of all bipartite unweighted graphs with parts A and B in which a_0 has an edge to all vertices in B and b_0 has an edge to every vertex in A. The purpose of nodes a_0 and b_0 is to keep all graphs in $\mathcal{G}$ connected. Bipartiteness implies that all graphs in $\mathcal{G}$ have girth 4 and $|\mathcal{G}| = 2^{q^2}$.

We will now bound the number of graphs in $\mathcal{G}$ that a single routing scheme can support. This will give a lower bound on the number of different routing schemes for $\mathcal{G}$.

Fix any routing scheme $\mathcal{R}$ and node labels $\mathcal{L}$. For any node $v \in A \setminus \{a_0\}$, consider the map $X : B \to [\deg(v)]$ defined by $X(u) = \big(R_v(l_u, 0)\big)_2$, i.e., the port number at v through which a message originating from v to u is sent. Note that if $\{v, u\} \in E$ then the port $X(u)$ at v should lead to u; otherwise, the path taken will have length ≥ 3 violating the stretch requirement. We will now use this constraint to limit the number of different graphs in $\mathcal{G}$ that can be supported by $\mathcal{R}$ and $\mathcal{L}$ assuming the port assignments $\mathcal{T}$ can be varied.

Define $M = \max\{X(u) : u \in B\}$. First, observe that $\deg(v) \geq M$ because if $\deg(v) < M$, then routing to u with $X(u) = M$ will fail. Also observe that for each $1 \leq i \leq \deg(v)$, routing to the node behind port i will necessarily send the message directly to that node: $X(\tau_v(i)) = i$. Otherwise, routing to that node will traverse a different node and violate the stretch requirement. Hence, $M = \deg(v)$ and $X(u)$ partitions all nodes in B based on the outgoing port number, each partition being non-empty.

For all $1 \leq i \leq M$, define s_i to be the size of i's partition: $s_i = \big|\{u \in B : X(u) = i\}\big|$. We know that $\sum_{i=1}^{M} s_i = |B| = q + 1$ and that $\tau_v(i)$ has s_i possible values. Therefore, the neighbors of v can take at most $\prod_{i=1}^{M} s_i$ possible values, or in other words, $\mathcal{N}(v)$ is one of at most $\prod_{i=1}^{M} s_i$ possible configurations. It is known that the geometric average of a set of non-negative numbers cannot exceed their arithmetic average. Hence, $\prod_{i=1}^{M} s_i \leq \left(\frac{q+1}{M}\right)^M$. Differentiating the logarithm with respect to M, we find that the value is maximized when $M = \frac{q+1}{e}$ and thus $\prod_{i=1}^{M} s_i \leq e^{\frac{q+1}{e}}$. Hence, the routing scheme $\mathcal{R}$ with labels $\mathcal{L}$ satisfies at most $e^{\frac{q+1}{e} \cdot q}$ graphs in $\mathcal{G}$. I.e., if for a graph G, routing scheme $\mathcal{R}$, labels $\mathcal{L}$, and port assignment $\mathcal{T}$ we define the predicate $\mathrm{Sat}(G, \mathcal{R}, \mathcal{L}, \mathcal{T})$ to be true if and only if $(\mathcal{R}, \mathcal{L}, \mathcal{T})$ satisfy G with routing stretch < 3, then

$$\Big|\big\{G \in \mathcal{G} : \exists \mathcal{T}, \mathrm{Sat}(G, \mathcal{R}, \mathcal{L}, \mathcal{T})\big\}\Big| \leq e^{\frac{q+1}{e} \cdot q}.$$

Since the labels have size at most $f(n)$ bits, the number of possible configurations of labels $\mathcal{L}$ is at most $(2^{f(n)+1} - 1)^n < 2^{(f(n)+1)n}$. Hence, the number of graphs in $\mathcal{G}$ that $\mathcal{R}$ can satisfy is

$$\Big|\big\{G \in \mathcal{G} : \exists (\mathcal{T}, \mathcal{L}), \mathrm{Sat}(G, \mathcal{R}, \mathcal{L}, \mathcal{T})\big\}\Big| < e^{\frac{q+1}{e} \cdot q} \cdot 2^{(f(n)+1)n}.$$

Therefore, the number of routing schemes necessary to satisfy each graph in $\mathcal{G}$ is larger than

$$\frac{|\mathcal{G}|}{e^{\frac{q+1}{e} \cdot q} \cdot 2^{(f(n)+1)n}} = \frac{2^{q^2}}{e^{\frac{q+1}{e} \cdot q} \cdot 2^{(f(n)+1)n}}.$$

Since

$$\log \frac{2^{q^2}}{e^{\frac{q+1}{e}\cdot q} \cdot 2^{(f(n)+1)n}} = q^2 - \frac{\log e}{e} q(q+1) - \big(f(n)+1\big)n =$$
$$\Big(\frac{n}{2}-1\Big)^2 - \frac{\log e}{2e} n\Big(\frac{n}{2}-1\Big) - \big(f(n)+1\big)n =$$
$$\Big(\frac{1}{4} - \frac{\log e}{4e}\Big)n^2 - \big(f(n)+1\big)n + \Big(\frac{\log e}{2e} - \frac{1}{2}\Big)n + 1 \in \Omega(n^2)$$

for an appropriate $f(n)$, one of the graphs in $\mathcal{G}$ requires $\Omega(n)$ bits of state at some node by lemma 1. ◀

A small modification of this argument lets us prove a slightly stronger statement, that any constant fraction of n nodes must have $\Omega(n)$ bits of state on some graph. Using the same family of graphs $\mathcal{G}$, simple counting manipulations, and the bound on the number of possible neighbor sets of a given node from above, we count the number of possible graphs possessing a routing scheme with too many small routing programs. We find that this number is less than $|\mathcal{G}|$ implying that at least one graph must not have too many small routing programs.

▶ **Theorem 3.** *For any $0 < c < 1$ there exist functions $f(n), g(n) \in \Omega(n)$ (f depends on c) such that if node labels have size $\leq f(n)$ bits, then there exists an unweighted graph with n nodes for which any routing scheme with stretch < 3 contains $\geq cn$ routing programs of size $\geq g(n)$ bits.*

Proof. Let $0 < c < 1$, $f(n) = \frac{1-c}{40}n$, $g(n) = \frac{n}{8}$, and define $k = \big\lfloor \frac{1-c}{2}n - 1 \big\rfloor$. We consider the same set of graphs $\mathcal{G}$ as in the previous theorem.

Fix node labels $\mathcal{L}$ and let $A' \subseteq A \setminus \{a_0\}$ such that $|A'| = k$. Now suppose for all $v \in A'$, $|R_v| < g(n)$, v's routing program is smaller than $g(n)$ bits. Then there are less than $2^{g(n)+1}$ different routing programs for each node in A'. By the argument in the previous theorem, each node in A' can have $< e^{\frac{q+1}{e}} \cdot 2^{g(n)+1}$ distinct sets of neighbors. Trivially, all other nodes in A can have at most 2^q distinct sets of neighbors. Therefore, the number of possible graphs is

$$\Big|\Big\{G \in \mathcal{G} : \exists(\mathcal{T}, \mathcal{R}), \big(\forall v \in A', |R_v| < g(n)\big) \wedge \mathrm{Sat}(G, \mathcal{R}, \mathcal{L}, \mathcal{T})\Big\}\Big| <$$
$$\big(e^{\frac{q+1}{e}} \cdot 2^{g(n)+1}\big)^k \cdot \big(2^q\big)^{q-k}.$$

Therefore, if A' can be arbitrary, the number of possible graphs is

$$\Big|\Big\{G \in \mathcal{G} : \exists(\mathcal{T}, \mathcal{R}), \mathrm{Sat}(G, \mathcal{R}, \mathcal{L}, \mathcal{T}) \wedge \big|\big\{v \in A \setminus \{a_0\} : |R_v| < g(n)\big\}\big| \geq k\Big\}\Big| =$$
$$\Big|\Big\{G \in \mathcal{G} : \exists(\mathcal{T}, \mathcal{R}, A'), \big(\forall v \in A', |R_v| < g(n)\big) \wedge \mathrm{Sat}(G, \mathcal{R}, \mathcal{L}, \mathcal{T})\Big\}\Big| <$$
$$\big(e^{\frac{q+1}{e}} \cdot 2^{g(n)+1}\big)^k \cdot \big(2^q\big)^{q-k} \cdot 2^q$$

(assuming $\exists A'$ means A' is taken from $A \setminus \{a_0\}$ such that $|A'| = k$).

If, in addition, the node labels can be varied, the number of possible graphs is

$$\Big|\Big\{G \in \mathcal{G} : \exists(\mathcal{T}, \mathcal{R}, \mathcal{L}), \mathrm{Sat}(G, \mathcal{R}, \mathcal{L}, \mathcal{T}) \wedge \big|\big\{v \in A \setminus \{a_0\} : |R_v| < g(n)\big\}\big| \geq k\Big\}\Big| <$$
$$\big(e^{\frac{q+1}{e}} \cdot 2^{g(n)+1}\big)^k \cdot \big(2^q\big)^{q-k} \cdot 2^q \cdot 2^{(f(n)+1)n}.$$

This is less than $2^{q^2}/2$ since

$$\log\Big(\big(e^{\frac{q+1}{e}}\cdot 2^{g(n)+1}\big)^k\cdot (2^q)^{q-k}\cdot 2^q\cdot 2^{(f(n)+1)n}\Big)=$$
$$k\Big(\frac{q+1}{e}\cdot\log e+g(n)+1\Big)+q(q-k)+q+(f(n)+1)n=$$
$$-k\Big(q-\frac{\log e}{e}(q+1)-g(n)-1\Big)+q^2+q+nf(n)+n=$$
$$-\Big\lfloor\frac{1-c}{2}n-1\Big\rfloor\Big(\frac{n}{2}-1-\frac{\log e}{2e}n-\frac{n}{8}-1\Big)+\frac{n}{2}\Big(\frac{n}{2}-1\Big)+\frac{1-c}{40}n^2+n<$$
$$-\Big(\frac{1-c}{2}n-2\Big)\Big(\frac{n}{10}-2\Big)+\frac{n}{2}\Big(\frac{n}{2}-1\Big)+\frac{1-c}{40}n^2+n=$$
$$\Big(\frac{1}{4}-\frac{1-c}{40}\Big)n^2+O(n)<\Big(\frac{n}{2}-1\Big)^2-1=q^2-1.$$

By symmetry, the number of possible graphs with a routing scheme having many small programs in $B\setminus\{b_0\}$

$$\Big|\Big\{G\in\mathcal{G}:\exists(\mathcal{T},\mathcal{R},\mathcal{L}),\mathrm{Sat}(G,\mathcal{R},\mathcal{L},\mathcal{T})\wedge\big|\big\{v\in B\setminus\{b_0\}:|R_v|<g(n)\big\}\big|\geq k\Big\}\Big|<2^{q^2}/2.$$

Hence, there are less than 2^{q^2} graphs which have a routing scheme with at least k small programs in $A\setminus\{a_0\}$ or $B\setminus\{b_0\}$:

$$\Big|\Big\{G\in\mathcal{G}:\exists(\mathcal{T},\mathcal{R},\mathcal{L}),\mathrm{Sat}(G,\mathcal{R},\mathcal{L},\mathcal{T})\wedge$$
$$\big(|\{v\in A\setminus\{a_0\}:|R_v|<g(n)\}|\geq k\vee|\{v\in B\setminus\{b_0\}:|R_v|<g(n)\}|\geq k\big)\Big\}\Big|<2^{q^2}.$$

So at least one graph G must have $<k$ small routing programs in both parts of the graph (excluding special vertices a_0 and b_0):

$$\forall(\mathcal{T},\mathcal{R},\mathcal{L}),\mathrm{Sat}(G,\mathcal{R},\mathcal{L},\mathcal{T})\rightarrow$$
$$\big|\big\{v\in A\setminus\{a_0\}:|R_v|<g(n)\big\}\big|<k\wedge\big|\big\{v\in B\setminus\{b_0\}:|R_v|<g(n)\big\}\big|<k.$$

It follows that the number of large programs must exceed $n-2-2k\geq cn$. ◀

We note that it's not true that *all* nodes must store $\Omega(n)$ bits of information. One can choose a node v and include in the label for each node u v's port toward u. This adds $O(\log n)$ bits to each node label and v can have constant size state.

3 Lower bounds with adversarial port numbers

3.1 Stretch <3

We will now prove an obvious fact which we haven't seen elsewhere: if the port assignment $\mathcal{T}$ are adversarially chosen and node labels are not, then some node must have $\Omega(n\log n)$ bits of state. Note the additional $\log n$ factor compared to the previous theorems. This implies that the trivial shortest paths routing scheme has optimal space up to a constant. Our main theorem in this section is strictly more general; however, this smaller result serves as a warmup and also lets us prove an $\Omega(n\log n)$ bit space lower bound for any constant fraction of nodes, similar to theorem 3.

▶ **Theorem 4.** *There exists a function $f(n)\in\Omega(n\log n)$ such that if node labels have size $\leq f(n)$ bits, then there exists an unweighted graph with n nodes for which any routing scheme with adversarial port assignment and stretch <3 contains a routing program of size $\Omega(n\log n)$ bits.*

Proof. Let $n = 2q$ (for odd n the proof is similar) and define $f(n) = \frac{\log(q!)}{2} \in \Omega(n \log n)$. We consider only one graph $G = K_{q,q}$, a full bipartite graph with parts of size q, and vary the port assignments $\mathcal{T}$. Let $\tilde{\mathcal{T}}$ be the set of all possible port assignments for G. Then $|\tilde{\mathcal{T}}| = (q!)^n$.

Because the stretch must be less than 3, routing from a node u in one part of G to a node v in the other part must traverse the edge $\{u, v\}$. Thus if we fix the node labels $\mathcal{L}$, every port assignment $\mathcal{T}$ requires a distinct routing scheme $\mathcal{R}$. Then if node labels $\mathcal{L}$ are varied, same routing scheme $\mathcal{R}$ can support at most $2^{(f(n)+1)n}$ port assignments:

$$\left|\left\{\mathcal{T} \in \tilde{\mathcal{T}} : \exists \mathcal{L}, \mathrm{Sat}(G, \mathcal{R}, \mathcal{L}, \mathcal{T})\right\}\right| \leq 2^{(f(n)+1)n}.$$

Then there must be at least

$$\frac{|\tilde{\mathcal{T}}|}{2^{(f(n)+1)n}} = \frac{(q!)^n}{2^{(\frac{\log(q!)}{2}+1)n}}$$

distinct routing schemes to support all possible port assignments.

Since

$$\log \frac{(q!)^n}{2^{(\frac{\log(q!)}{2}+1)n}} = n \log(q!) - \left(\frac{\log(q!)}{2} + 1\right)n \in \Omega(n^2 \log n),$$

some routing program must have size $\Omega(n \log n)$ bits. ◀

Using the techniques in the proof of theorem 3, one can prove this space lower bound for any constant fraction of nodes.

▶ **Theorem 5.** *For any $0 < c < 1$ there exist functions $f(n), g(n) \in \Omega(n \log n)$ (f depends on c) such that if node labels have size $\leq f(n)$ bits, then there exists an unweighted graph with n nodes for which any routing scheme with adversarial port assignment and stretch < 3 contains $\geq cn$ routing programs of size $\geq g(n)$ bits.*

3.2 Stretch $< 2k + 1$

We would like to show for any integer $k \geq 1$ that when port assignment is adversarial, the worst case per-node space complexity of a routing scheme of stretch $< 2k + 1$ is $\Omega\big(n^{\frac{1}{k}} \log n\big)$. We are able to show this only assuming a well-known girth conjecture by Erdős.

▶ **Definition 6.** *We say that an unweighted undirected graph G has girth s if and only if all cycles in G have length $\geq s$.*

▶ **Definition 7.** *For any integers $s, n \geq 1$, define $M_s(n)$ to be the largest m such that there exists a graph with n vertices, m edges and girth s.*

It is known that any graph with n nodes and $n^{1+\frac{1}{k}}$ edges must have a cycle of length at most $2k$. So, $M_{2k+2}(n) \in O\big(n^{1+\frac{1}{k}}\big)$. However, $M_{2k+2}(n) \in \Omega\big(n^{1+\frac{1}{k}}\big)$ has only been proven for $k = 1, 2, 3, 5$; for other k it remains an open question. Also, $M_{2k+1}(n) \in \Theta\big(M_{2k+2}(n)\big)$ since any graph with n vertices, m edges and girth $2k + 1$ must contain a bipartite subgraph with $\geq \frac{m}{2}$ edges (which necessarily has girth $2k + 2$). For an overview of the best known lower bounds for $M_s(n)$, refer to [16]. We note that most constructions of large graphs with some fixed girth rely on finite fields, and the number of vertices n is thus required to be some polynomial of a prime or a prime power. To prove a lower bound for all sufficiently large n, and thus justify the use of Ω, one can employ Bertrand's postulate which states that for all

natural numbers i, $p_{i+1} < 2p_i$, where p_i is the i-th prime number. Justification of Ω easily follows from here: given an integer n, we construct a dense large girth graph with p_i vertices where $\frac{n}{2} < p_i \leq n$, and add missing $n - p_i$ vertices along with $n - p_i$ edges keeping the final graph connected.

Having a graph G with n vertices and $\Omega\left(n^{1+\frac{1}{k}}\right)$ edges lets us show that there must be at least $2^{\Omega(n^{1+\frac{1}{k}} \log n)}$ routing schemes to satisfy all possible node labels $\mathcal{T}$ for G. Hence, at least some node must use $\Omega\left(n^{\frac{1}{k}} \log n\right)$ bits of space in the worst case. Weaker lower bounds for $M_{2k+2}(n)$ give corresponding weaker results.

▶ **Theorem 8.** *Let $k \geq 1$ be an integer constant. If port numbers are adversarially chosen, $\frac{m}{n}$ is sufficiently large, there exists a graph with n vertices, m edges and girth $2k+2$, and all node labels have size $\leq L = \frac{1}{16k} \cdot \log \frac{m}{n} \cdot \frac{m}{n} - 1$, then there also exists a port assignment $\mathcal{T}$ for G such that any routing scheme for $(G, \mathcal{T})$ of stretch $< 2k+1$ requires $\frac{1}{32k} \cdot \log \frac{m}{n} \cdot \frac{m}{n}$ bits of space at some node.*

Proof. Let $G = (V, E)$ be a graph with n vertices, m edges and girth $2k+2$, where $V = [n]$. For any pair of neighbors $\{x, y\} \in E$, the route from x to y chosen by a routing scheme of stretch $< 2k+1$ must be of length $\leq 2k$. Therefore, during routing from x to y, x must send the packet to y, most likely as the last step; otherwise, there would be a cycle in G of length $\leq 2k+1$ which violates the girth requirement. This doesn't, however, mean that routing from x to y must take the shortest path. x might send the message to some other neighbor z which returns the message to x (with a modified header) which now sends the message to y. In general, if you trace the message from x to y, the visited nodes and traversed edges form an arbitrary tree (see figure 1).

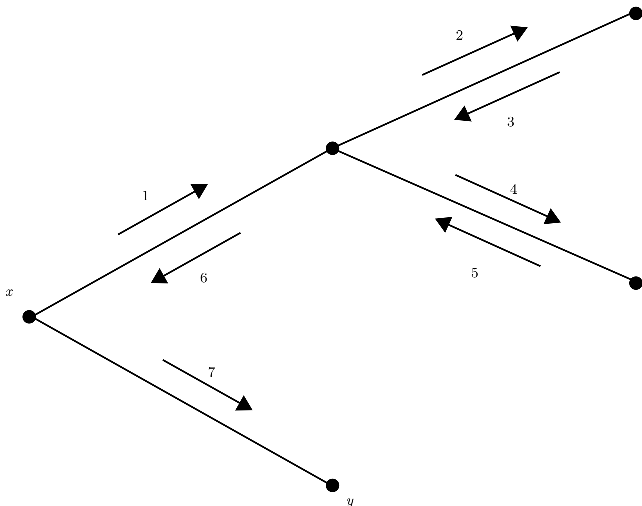


Figure 1 Routing with stretch $< 2k+1$ in a graph with girth $2k+2$.

We consider only a single graph G and all possible port assignments $\mathcal{T}$ for G. Define $m_i = \deg(i)$. Then the number of all possible port assignments for G is $\prod_{i=1}^{n} m_i!$. As in theorem 2, we will argue that any routing scheme cannot satisfy many distinct port assignments.

Fix the node labels $\mathcal{L}$ (as the proof of theorem 2 shows, they do not have much of an effect) and a routing scheme $\mathcal{R}$. Here is the main idea of the proof. Take any pair of neighbors $\{x, y\} \in E$. Then the port $p = R_x(y, 0)_2$ through which x will send the message toward y is

determined. The neighbor behind port p can be arbitrary, but if we fix it, say z, and we fix z's port to x, then the next hop is determined and the next out port through which z will forward the message will also be determined. We can keep tracing the packet through the network fixing the ports along the way. Eventually, however, node x must send the packet to node y, and this is where we can "extract" a port assignment from the routing programs. Hence, the routing scheme must "store" approximately every $4k$-th port assignment.

To formalize this argument, we describe a program that, given node labels $\mathcal{L}$, a routing scheme $\mathcal{R}$, port assignment on vertices with small degrees $S = \left\{x \in V : m_x < \frac{m}{n}\right\}$, and a bit string, can recover the port assignment $\mathcal{T}$ on all vertices. The program will iteratively pick neighbor vertices x and y such that the port at x toward y is undefined, and try to simulate the routing as in the example above reading the number of hops until the edge $\{x, y\}$ is traversed and unknown port assignments along the path from the bit string. We will then argue that this bit string does not need to be long, which intuitively implies that the routing scheme $\mathcal{R}$ must contain much information.

The program will work with a list of partial functions, one for each vertex, signifying which ports have already been fixed. Denote the set of all possible such lists by $\mathcal{P}$, and define $\vec{P} \in \mathcal{P}$ if and only if $\vec{P} = (P_1, P_2, ..., P_n)$ where each $P_i \subseteq [m_i] \times \mathcal{N}(i)$ such that for each $p \in [m_i]$ there is at most one j such that $(p, j) \in P_i$ and for each j there is at most one p such that $(p, j) \in P_i$.

We now formally describe the program which has access to $\mathcal{L} = \{l_i\}_{i\in[n]}$, $\mathcal{R} = \{R_i\}_{i\in[n]}$ and $\vec{P}$ such that $\forall i \in S, |\vec{P}_i| = m_i$ and $\forall i \in V \setminus S, |\vec{P}_i| = 0$, and reads bits from the input bit string.

while $\exists x \in [n], |\vec{P}_x| < m_x$ **do**
  $X \leftarrow \{i \in [n] : |\vec{P}_i| < m_i\}$
  $M \leftarrow \max\{m_i : i \in X\}$
  $x \leftarrow \min\{i \in X : m_i = M\}$
  $y \leftarrow \min\{i \in \mathcal{N}(x) : (\vec{P}_x)^{-1}(i)$ is undefined$\}$
  $r \leftarrow$ read $\lceil \log(2k-1) \rceil$-bit number ▷ read the number of hops
  $h \leftarrow l_y$
  $p_{\text{in}} \leftarrow 0$
  for r times **do**
    $p_{\text{out}} \leftarrow R_x(h, p_{\text{in}})_2$
    $h \leftarrow R_x(h, p_{\text{in}})_1$
    if $\vec{P}_x(p_{\text{out}})$ is undefined **then**
      $i \leftarrow$ read $\lceil \log m_x \rceil$-bit number ▷ read the "ID" of the neighbor behind port p_{out}
      $\vec{P}_x \leftarrow \vec{P}_x \cup \{(p_{\text{out}}, \mathcal{N}(x)(i))\}$ ▷ update the known port assignments
    end if
    $z \leftarrow \vec{P}_x(p_{\text{out}})$ ▷ next visited node
    if $(\vec{P}_z)^{-1}(x)$ is undefined **then**
      $p \leftarrow$ read $\lceil \log m_z \rceil$-bit number ▷ read the port we are entering through at z
      $\vec{P}_z \leftarrow \vec{P}_z \cup \{(p, x)\}$
    end if
    $p_{\text{in}} \leftarrow (\vec{P}_z)^{-1}(x)$
    $x \leftarrow z$
  end for
  $p_{\text{out}} \leftarrow R_x(h, p_{\text{in}})_2$
  $\vec{P}_x \leftarrow \vec{P}_x \cup (p_{\text{out}}, y)$ ▷ learn this port assignment "for free"
end while
return $\vec{P}$

We will now bound the length of the bit string needed for this program to run correctly. The following outer loop invariant is claimed: if b is the number of bits read so far, then

$$b \leq \sum_{i \in V \setminus S} |\vec{P_i}| \cdot \Big(\lceil \log(2k) \rceil + \big(1 - \frac{1}{4k}\big) \lceil \log m_i \rceil \Big).$$

It is correct before the loop starts since $b = 0$. Assume it is correct after some number of iterations, and let b' be the number of bits read and $\vec{P}'$ be the updated port assignment at the end of the iteration. We need to prove that

$$b' \leq \sum_{i \in V \setminus S} |\vec{P}'_i| \cdot \Big(\lceil \log(2k) \rceil + \big(1 - \frac{1}{4k}\big) \lceil \log m_i \rceil \Big).$$

Let t be the number of individual port assignments discovered through reading from the bit string, and let $z_1, ..., z_t$ be the nodes that own those ports. Notice that $t \leq 4k - 2$ since after $4k - 2$ traversed ports, the next port must lead from x directly to y and we do not read it from the bit string. Also notice that the degrees of the nodes $z_1, ..., z_t$ do not exceed that of x, the source node. This is because the algorithm picks x with the largest degree among those that have free ports. Then

$$\sum_{i \in V \setminus S} |\vec{P}'_i| \cdot \Big(\lceil \log(2k) \rceil + \big(1 - \frac{1}{4k}\big) \lceil \log m_i \rceil \Big) =$$

$$\sum_{i \in V \setminus S} |\vec{P_i}| \cdot \Big(\lceil \log(2k) \rceil + \big(1 - \frac{1}{4k}\big) \lceil \log m_i \rceil \Big) + \sum_{i=1}^{t} \Big(\lceil \log(2k) \rceil + \big(1 - \frac{1}{4k}\big) \lceil \log m_{z_i} \rceil \Big) +$$

$$\Big(\lceil \log(2k) \rceil + \big(1 - \frac{1}{4k}\big) \lceil \log m_x \rceil \Big) \geq$$

$$b + \lceil \log(2k) \rceil + \big(1 - \frac{1}{4k}\big) \Big(\sum_{i=1}^{t} \lceil \log m_{z_i} \rceil + \lceil \log m_x \rceil \Big) =$$

$$b + \lceil \log(2k) \rceil + \sum_{i=1}^{t} \lceil \log m_{z_i} \rceil + \lceil \log m_x \rceil - \frac{1}{4k} \Big(\sum_{i=1}^{t} \lceil \log m_{z_i} \rceil + \lceil \log m_x \rceil \Big) [\geq]$$

Since in this iteration of the outer loop we read $\lceil \log(2k-1) \rceil + \sum_{i=1}^{t} \lceil \log m_{z_i} \rceil$ bits,

$$[\geq] b' + \lceil \log m_x \rceil - \frac{1}{4k} \Big(\sum_{i=1}^{t} \lceil \log m_{z_i} \rceil + \lceil \log m_x \rceil \Big) \geq$$

$$b' + \lceil \log m_x \rceil - \frac{1}{4k} \big((4k-2) \lceil \log m_x \rceil + \lceil \log m_x \rceil \big) \geq b'.$$

Hence, the loop invariant holds and the algorithm reads at most a total of B bits where

$$B = \sum_{i \in V \setminus S} m_i \big(\lceil \log(2k) \rceil + (1 - \frac{1}{4k}) \lceil \log m_i \rceil \big) \leq \sum_{i \in V \setminus S} m_i \big((1 - \frac{1}{4k}) \log m_i + \log k + 3 \big).$$

Since node labels have size $\leq L = \frac{1}{16k} \cdot \log \frac{m}{n} \cdot \frac{m}{n} - 1$, the number of all possible $\mathcal{L}$ is $\big(2^{L+1} - 1\big)^n \leq 2^{(L+1)n} = 2^{\frac{1}{16k} \cdot \log \frac{m}{n} \cdot m}$. Then, defining $\mathrm{Sat}(G, \mathcal{R}, \mathcal{L}, \mathcal{T})$ true if and only if $\mathcal{R}$ is a correct routing scheme with stretch $< 2k + 1$ for graph G with labels $\mathcal{L}$ and port assignment $\mathcal{T}$,

$$\Big| \big\{ \mathcal{T} : \exists \mathcal{L}, \mathrm{Sat}(G, \mathcal{R}, \mathcal{L}, \mathcal{T}) \big\} \Big| \leq \qquad 2^{\frac{1}{16k} \cdot \log \frac{m}{n} \cdot m} \cdot \prod_{i \in S} m_i! \cdot 2^B.$$

Therefore, the number of routing schemes necessary to satisfy all port assignments is at least

$$\frac{\prod_{i=1}^{n} m_i!}{2^{\frac{1}{16k}\cdot\log\frac{m}{n}\cdot m}\cdot\prod_{i\in S} m_i!\cdot 2^B}.$$

We now calculate the logarithm of this expression.

$$\log\frac{\prod_{i=1}^{n} m_i!}{2^{\frac{1}{16k}\cdot\log\frac{m}{n}\cdot m}\cdot\prod_{i\in S} m_i!\cdot 2^B} = \log\frac{\prod_{i\in V\setminus S} m_i!}{2^{\frac{1}{16k}\cdot\log\frac{m}{n}\cdot m}\cdot 2^B} =$$

$$\sum_{i\in V\setminus S}\log\left(m_i!\right) - \frac{1}{16k}\cdot\log\frac{m}{n}\cdot m - B[\geq]$$

Using Stirling's approximation,

$$[\geq]\sum_{i\in V\setminus S}(m_i\log m_i - cm_i) - \frac{1}{16k}\cdot\log\frac{m}{n}\cdot m - B =$$

$$\sum_{i\in V\setminus S}(m_i\log m_i - cm_i) - \frac{1}{16k}\cdot\log\frac{m}{n}\cdot m - \sum_{i\in V\setminus S} m_i\Big((1-\frac{1}{4k})\log m_i + \log k + 3\Big) =$$

$$\sum_{i\in V\setminus S} m_i\Big(\log m_i - c - (1-\frac{1}{4k})\log m_i - \log k - 3\Big) - \frac{1}{16k}\cdot\log\frac{m}{n}\cdot m =$$

$$\sum_{i\in V\setminus S} m_i(\frac{1}{4k}\log m_i - c - \log k - 3) - \frac{1}{16k}\cdot\log\frac{m}{n}\cdot m[\geq]$$

If $i\in V\setminus S$, then $m_i\geq\frac{m}{n}$. Thus,

$$[\geq]\sum_{i\in V\setminus S} m_i(\frac{1}{4k}\log\frac{m}{n} - c - \log k - 3) - \frac{1}{16k}\cdot\log\frac{m}{n}\cdot m[\geq]$$

Assuming $\frac{m}{n}$ is sufficiently large,

$$[\geq]\sum_{i\in V\setminus S}\frac{1}{8k}m_i\log\frac{m}{n} - \frac{1}{16k}\cdot\log\frac{m}{n}\cdot m =$$

$$\frac{1}{8k}\cdot\log\frac{m}{n}\cdot\sum_{i\in V\setminus S} m_i - \frac{1}{16k}\cdot\log\frac{m}{n}\cdot m =$$

$$\frac{1}{8k}\cdot\log\frac{m}{n}\cdot\Big(2m - \sum_{i\in S} m_i\Big) - \frac{1}{16k}\cdot\log\frac{m}{n}\cdot m \geq$$

$$\frac{1}{8k}\cdot\log\frac{m}{n}\cdot\Big(2m - n\cdot\frac{m}{n}\Big) - \frac{1}{16k}\cdot\log\frac{m}{n}\cdot m \geq$$

$$\frac{1}{8k}\cdot\log\frac{m}{n}\cdot m - \frac{1}{16k}\cdot\log\frac{m}{n}\cdot m =$$

$$\frac{1}{16k}\cdot\log\frac{m}{n}\cdot m.$$

By lemma 1, there is a port assignment on which some node must use $\frac{1}{16k}\cdot\log\frac{m}{n}\cdot\frac{m}{n} - 1 \geq \frac{1}{32k}\cdot\log\frac{m}{n}\cdot\frac{m}{n}$ bits of space. ◀

▶ **Corollary 9.** *Let $k\geq 1$ be an integer constant. If $M_{2k+2}(n)\in\Omega(n^{1+\frac{1}{k}})$ and node labels have size $\leq\Omega(n^{\frac{1}{k}}\log n)$, then for any n there is a graph G on n nodes and a port assignment $\mathcal{T}$ for G such that any routing scheme for $(G,\mathcal{T})$ of stretch $< 2k+1$ requires $\Omega(n^{\frac{1}{k}}\log n)$ bits of space at some node.*

4 Known techniques require girth conjecture

All known proofs of lower bounds for space complexity of non-shortest path routing schemes in the labeled setting, such as the ones in [14], [10] or this paper, consist of proving that a large number of routing schemes are necessary to satisfy all possible graphs and port assignments (if adversarially chosen). Then it follows that some graph and port assignment requires a large number of bits stored at some node.

We note that proving that a large number of routing schemes is necessary must require existence of graphs with large girth and many edges. Therefore, if one wishes to not use a girth conjecture, some other technique, such as the one in [2] for the name-independent setting, is needed.

We utilize spanners with low stretch and large girth.

▶ **Definition 10.** *Let G be a graph with n vertices. We call a subgraph S of G a t-spanner if and only if for any two vertices u, v connected in G, $d_S(u, v) \le t \cdot d_G(u, v)$ (distance in S is stretched at most t times compared to the distance in G).*

The following lemma is proved in [6], but we provide the proof for completeness.

▶ **Lemma 11.** *Let G be a graph with n vertices. There exists a t-spanner S of G with girth $t + 2$.*

Proof. We construct a spanner S similarly to Kruskal's minimum spanning tree algorithm. Go through all edges in G, and for each edge $\{u, v\}$, add it to S if and only if there isn't a path between u and v in S of length $\le t$.

Clearly, at the end of the algorithm, for any edge $\{u, v\}$ in G, $d_S(u, v) \le t$. Therefore, S is a t-spanner.

We only need to show that S has girth $t + 2$. Suppose during the algorithm we add an edge and create a cycle of length $l \le t + 1$. Then there was already a path between the endpoints of length $l - 1 \le t$, and we shouldn't add an edge. ◀

▶ **Theorem 12.** *Suppose $n \ge 2$ and $s \ge 2$ are integers, and N routing schemes are necessary to satisfy all graphs with n vertices and all port assignments with stretch $< s$. Then $M_{s+1}(n) \ge \frac{\log N}{4 \log n} - 1$; i.e., there exists a graph with $\ge \frac{\log N}{4 \log n} - 1$ edges and girth $s + 1$.*

Proof. Suppose $M_{s+1}(n) < \frac{\log N}{4 \log n} - 1$; i.e., all graphs with girth $s + 1$ have $< \frac{\log N}{4 \log n} - 1$ edges. We want to show that then less than N routing schemes are sufficient to satisfy all graphs and port assignments.

Given a graph G and a port assignment $\mathcal{T}$, we simply encode in each node's routing program its ID, an $(s - 1)$-spanner S of G with girth $s + 1$ which exists by lemma 11, and the port assignment $\mathcal{T}$. All this information allows for stretch $s - 1$ routing.

It is easy to see that there are at most $(n^2)^m$ graphs with n vertices and m edges. Also, for each such graph there are at most $(n^2)^m$ port assignments. Therefore, the number of such routing schemes is at most

$$\sum_{i=0}^{M_{s+1}(n)} \left(n^2\right)^i \cdot \left(n^2\right)^i = \sum_{i=0}^{M_{s+1}(n)} \left(n^4\right)^i = \frac{\left(n^4\right)^{M_{s+1}(n)+1} - 1}{n^4 - 1} < \left(n^4\right)^{M_{s+1}(n)+1}.$$

Since $\log\left(\left(n^4\right)^{M_{s+1}(n)+1}\right) = 4\big(M_{s+1}(n)+1\big)\log n < 4\left(\frac{\log N}{4 \log n}\right)\log n = \log N$, less than N routing schemes can satisfy all graphs with n vertices and all port assignments with stretch $< s$. This is a contradiction. ◀

▶ **Corollary 13.** *Let $k \geq 1$ be an integer constant. If $2^{\Omega(n^{1+\frac{1}{k}} \log n)}$ routing schemes are necessary to satisfy all graphs with n vertices and all port assignments with stretch $< 2k+1$, then $M_{2k+2} \in \Omega\big(n^{1+\frac{1}{k}}\big)$.*

References

1. Ittai Abraham, Cyril Gavoille, and Dahlia Malkhi. Routing with improved communication-space trade-off. In Rachid Guerraoui, editor, *Distributed Computing*, pages 305–319, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. doi:10.1007/978-3-540-30186-8_22.
2. Ittai Abraham, Cyril Gavoille, and Dahlia Malkhi. On space-stretch trade-offs: Lower bounds. In *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '06, pages 207–216, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1148109.1148143.
3. Ittai Abraham, Cyril Gavoille, and Dahlia Malkhi. On space-stretch trade-offs: Upper bounds. In *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '06, pages 217–224, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1148109.1148144.
4. Ittai Abraham, Cyril Gavoille, Dahlia Malkhi, Noam Nisan, and Mikkel Thorup. Compact name-independent routing with minimum stretch. *ACM Trans. Algorithms*, 4(3), July 2008. doi:10.1145/1367064.1367077.
5. Ittai Abraham and Dahlia Malkhi. Name independent routing for growth bounded networks. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 49–55, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1073970.1073978.
6. Ingo Althöfer, Gautam Das, David Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discrete Comput. Geom.*, 9(1):81–100, December 1993. doi:10.1007/BF02189308.
7. Harry Buhrman, Jaap-Henk Hoepman, and Paul Vitányi. Optimal routing tables. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 134–142, New York, NY, USA, 1996. Association for Computing Machinery. doi:10.1145/248052.248076.
8. Shiri Chechik. Compact routing schemes with improved stretch. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 33–41, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2484239.2484268.
9. Cyril Gavoille and Marc Gengler. Space-efficiency for routing schemes of stretch factor three. Technical report, Laboratoire Bordelais de Recherche en Informatique, Université Bordeaux, 1997. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.13.5857&rep=rep1&type=pdf.
10. Cyril Gavoille and Marc Gengler. Space-efficiency for routing schemes of stretch factor three. *Journal of Parallel and Distributed Computing*, 61(5):679–687, 2001. doi:10.1006/jpdc.2000.1705.
11. Cyril Gavoille and Stéphane Pérennès. Memory requirement for routing in distributed networks. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 125–133, New York, NY, USA, 1996. Association for Computing Machinery. doi:10.1145/248052.248075.
12. Goran Konjevod, Andréa W. Richa, and Donglin Xia. Scale-free compact routing schemes in networks of low doubling dimension. *ACM Trans. Algorithms*, 12(3), June 2016. doi:10.1145/2876055.
13. J. Matousek. On the distortion required for embedding finite metric spaces into normed spaces. *Israel Journal of Mathematics*, 93:333–344, 1996. doi:10.1007/BF02761110.
14. David Peleg and Eli Upfal. A trade-off between space and efficiency for routing tables. *J. ACM*, 36(3):510–530, July 1989. doi:10.1145/65950.65953.

15 Liam Roditty and Roei Tov. *New Routing Techniques and Their Applications*, pages 23–32. Association for Computing Machinery, New York, NY, USA, 2015. doi:10.1145/2767386.2767409.

16 Mikkel Thorup and Uri Zwick. Approximate distance oracles. In Jeffrey Scott Vitter, Paul G. Spirakis, and Mihalis Yannakakis, editors, *Proceedings on 33rd Annual ACM Symposium on Theory of Computing, July 6-8, 2001, Heraklion, Crete, Greece*, pages 183–192. ACM, 2001. doi:10.1145/380752.380798.

17 Mikkel Thorup and Uri Zwick. Compact routing schemes. In Arnold L. Rosenberg, editor, *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 2001, Heraklion, Crete Island, Greece, July 4-6, 2001*, pages 1–10. ACM, 2001. doi:10.1145/378580.378581.

Brief Announcement: Authenticated Consensus in Synchronous Systems with Mixed Faults

Ittai Abraham
VMware Research, Herzliya, Israel

Danny Dolev
The Hebrew University of Jerusalem, Israel

Alon Kagan
The Hebrew University of Jerusalem, Israel

Gilad Stern
The Hebrew University of Jerusalem, Israel

Abstract

Protocols solving authenticated consensus in synchronous networks with Byzantine faults have been widely researched and known to exists if and only if $n > 2f$ for f Byzantine faults. Similarly, protocols solving authenticated consensus in partially synchronous networks are known to exist if $n > 3f + 2k$ for f Byzantine faults and k crash faults. In this work we fill a natural gap in our knowledge by presenting MixSync, an authenticated consensus protocol in synchronous networks resilient to f Byzantine faults and k crash faults if $n > 2f + k$. As a basic building block, we first define and then construct a publicly verifiable crusader agreement protocol with the same resilience. The protocol uses a simple double-send round to guarantee non-equivocation, a technique later used in the MixSync protocol. We then discuss how to construct a state machine replication protocol using these ideas, and how they can be used in general to make such protocols resilient to crash faults. Finally, we prove lower bounds showing that $n > 2f + k$ is optimally resilient for consensus and state machine replication protocols.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases consensus, state machine replication, mixed faults, synchrony, lower bounds

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.38

Related Version *Full Version*: https://eprint.iacr.org/2022/805

Funding This work was supported in part by the HUJI Federmann Cyber Security Research Center in conjunction with the Israel National Cyber Directorate (INCD) in the Prime Minister's Office.

1 Introduction

In recent years there has been a surge of interest in Byzantine Fault Tolerance (BFT) and Blockchain technologies. The security of both Bitcoin and later Ethereum's proof-of-work protocols depends on a synchronous model and obtains resilience against minority corruptions [1, 2]. Following this direction there have been several academic papers that advanced *authenticated* BFT protocols and systems in the synchronous model that use more traditional membership assumptions. A major advantage of this model is that it can obtain resilience as long as $n > 2f$ which is qualitatively much better than protocols that assume partial synchrony (or asynchrony) that can only obtain resilience of $n > 3f$.

In this paper we continue this line of research into authenticated BFT protocols and merge it with yet another long line of research around *mixed-faults*. In the mixed-faults model that we study in this paper, the adversary can corrupt up to f parties in a malicious manner and can crash up to k additional parties. One motivation behind this assumption is that it allows us to model a real world case where the non-faulty parties can detect some of

© Ittai Abraham, Danny Dolev, Alon Kagan, and Gilad Stern;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 38; pp. 38:1–38:3
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the corrupted parties (via some side broadcast channel, say the internet or a secure messaging application) and publicly mark them as faulty, hence simulate a crash. Another motivation is to model the case that there is some trusted hardware that may cause some of the parties to crash if they become compromised. Another motivation, for a large set of parties, is that some of the honest parties may be offline for large periods of time, and hence can be modeled as crashed. Finally, there may be a need to model crash failures (due to hardware failures) as a separate parameter from Byzantine failures (due to an adversary).

To the best of our knowledge, this problem of authenticated BFT in synchrony with mixed-faults has not been systematically studied. It is well known that the best one can hope for in partial synchrony (or asynchrony) is security when $n > 3f + 2k$. This is where authenticated BFT in synchrony gives a major resilience advantage. The main result of this paper is that it is possible to get security *if and only if* $n > 2f + k$. We note that we do not find this bound very surprising, but we believe getting to this bound and proving tight upper and lower bounds provides new insights into how to design authenticated BFT protocols in the synchronous model.

Our Contributions – Upper Bounds. The main contribution of our paper is MixSync, an authenticated consensus protocol in a synchronous network resilient to f Byzantine faults and k crash faults if $n > 2f + k$. MixSync uses a simple technique for achieving authenticated non-equivocation in synchronous networks with mixed-faults. As far as we know, this is the first authenticated consensus protocol in a mixed-faults setting achieving a resilience of $n > 2f + k$ without limiting the power of the adversary. This is made possible by solving the task in a synchronous setting, as opposed to the partially synchronous setting which requires at least $n > 3f + 2k$ replicas. The protocol is oblivious to the number of crash faults, so it is possible to use it as long as some bound is known on the number of Byzantine replicas and at least $f + 1$ honest replicas are guaranteed to stay online.

To construct our new authenticated consensus protocol we decompose it into an outer protocol and an inner building block. We call this inner building block *Publicly Verifiable Crusader Agreement* (PVCA). We show how to construct a simple and efficient $PVCA$ protocol in a network with mixed faults. In this task, there is a commonly known sender with an input x, and replicas are required to output some value v and a proof π. If the sender is honest, every honest replica that completes the protocol outputs x and a proof π, showing that it is their actual output from the protocol. If the sender is faulty, then there exists some value v such that every honest replica either outputs v or $\perp$ with an appropriate proof. Using these proofs, replicas can convince each other that the value they received is correct, or alternatively that the sender was faulty by producing a proof for $\perp$. This task formalizes a rather strong notion of a non-equivocation round. By the end of the round, every replica that hears a message from the sender, hears the same message, in addition to proving that a given value was actually received from the sender (or that the sender was faulty). The $PVCA$ protocol relies on a simple technique: forwarding a received message to all replicas, and then sending a second message immediately after that. A replica receiving the second message knows that if its sender was non-Byzantine, the first message has already been sent to all replicas.

Using the simple idea of a double-send, we then construct the MixSync protocol. The protocol is based on the Sync HotStuff protocol, with slight adaptations made for it to solve the task of single-shot consensus. Our protocol is view and leader based, and just like Sync HotStuff, each view consists of a view change and a non-equivocation round. In order to make our protocol resilient to k crash faults, all that is needed is using the double-send

technique from our $PVCA$ protocol. In fact, we could use the $PVCA$ protocol as a blackbox inside the Sync HotStuff protocol, only requiring the addition of a view-change protocol, but we open the blackbox in order to optimize the protocol. Finally, we discuss how to create a State Machine Replication (SMR) protocol using the same double-send idea. This can either be done generically by using our consensus protocol, or by adapting optimized protocols such the Sync HotStuff protocol. Thankfully, in the Sync HotStuff protocol replicas already send two messages to all replicas after receiving a value from the leader, meaning that the only change required is making sure that replicas send them in a specific order.

These constructions suggest a possible general approach to constructing consensus and SMR protocols in the authenticated synchronous mixed-fault scenario. If a protocol mainly consists of a non-equivocation round and a view change protocol, replace the non-equivocation round with our $PVCA$ or simply a double-send, yielding a crash-resilient protocol. Specific protocols might also require adapting other parts of the protocol if they rely on specific properties of the non-equivocation round.

Our Contributions – Lower Bounds. In order to complete the picture, we also provide tight lower bounds for consensus tasks in the presence of mixed-faults in the synchronous model. First, we show that consensus is impossible in a system with f Byzantine faults and k crash faults if $n \leq 2f + k$. Secondly, in order to prove a similar lower bound for the task of SMR, we first formalize the notion of a Write-Once Register. A Write-Once Register is a shared memory object to which clients can write only once. That means that once a client manages to write a value to the register, this value is final and cannot be changed. Unlike registers that allow clients to overwrite previous values, the Write-Once Register captures a specific idea of finality that is shared with consensus protocols. In SMR protocols, replicas are required to commit to a value v_s for each log position $s \in \mathbb{N}$, which clients can read by asking replicas to send their committed values. This means that SMR protocols actually implement infinitely many Write-Once Registers. As such, we show that no protocol virtualizing a Write-Once Register exists in a system with f Byzantine faults and k crash faults if $n \leq 2f + k$, yielding a lower bound on SMR protocols as well.

References

1 Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 281–310. Springer, 2015.

2 Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 643–673. Springer, 2017.

Brief Announcement: It's not easy to relax: liveness in chained BFT protocols

Ittai Abraham ✉
VMware Research, Herzliya, Israel

Natacha Crooks ✉
UC Berkeley, CA, USA

Neil Giridharan[1] ✉
UC Berkeley, CA, USA

Heidi Howard ✉
Microsoft Research Cambridge, Cambridge, UK

Florian Suri-Payer ✉
Cornell University, Ithaca, NY, USA

Abstract

Modern *chained* BFT SMR protocols have poor liveness under failures as they require multiple consecutive honest leaders to commit a single block. Siesta, our proposed new BFT SMR protocol, is instead able to commit a block that spans multiple non-consecutive honest leaders. Siesta reduces the expected commit latency of HotStuff by a factor of three under failures, and the worst-case latency by a factor of eight.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Consensus, blockchain, BFT

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.39

Related Version *Full Version*: https://arxiv.org/abs/2205.11652 [1]

1 Introduction

Blockchain systems add two constraints over traditional BFT SMR protocols: 1) only valid operations should be committed and 2) operation ordering should be fair. To address these concerns, recent blockchain protocols such as Casper FFG [2], HotStuff [5], DiemBFTv4 [4], and Fast-Hotstuff [3] are structured around two key building blocks: Chaining and Leader-Speaks-Once (LSO).

Chaining. To ensure validity, existing protocols evaluate proposed blocks against the full sequential history of the chain and check application-level preconditions are satisfied. Most BFT protocols require a (worst-case) minimum of two voting rounds (henceforth *phases*). Each voting phase aims to establish a *quorum certificate (QC)* by collecting a set of signed votes from a majority of honest replicas. Blockchain systems *pipeline* the voting phases of consecutive proposals to avoid redundant coordination and cryptography: the system can use the quorum certificate of the second voting phase of block i to certify the first phase of block $i+1$. Each block requires (on average) generating and verifying the signatures of a single QC. This is especially important for large participant sets as QC sizes grow linearly with the number of replicas, increasing cryptographic costs.

[1] Lead author.

© Ittai Abraham, Natacha Crooks, Neil Giridharan, Heidi Howard, and Florian Suri-Payer; licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 39; pp. 39:1–39:3
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Leader-Speaks-Once (LSO). To minimize fairness concerns associated with leader-based solutions and to decrease the influence of adaptive adversaries (adversaries who control the network), BFT protocols targeted at blockchains adopt a *leader-speak-once* (LSO) model. In LSO, each leader proposes and certifies a single block after which the leader is immediately rotated out as part of a new *view*. Electing a different leader per block limits the leader's influence; it can manipulate transactions in the proposed block only. Traditional BFT protocols, in contrast, adopt a *stable-leader* paradigm in which leaders are only replaced if they fail to make progress through *view change*. View changes are intentionally costly procedures and assumed to be infrequent. Complex view changes allow for a simpler and more efficient failure-free steady state. LSO protocols instead perform view changes proactively in each round, and thus are required to place the view change on the critical path of the system.

While a joint chained leader-speak-once (CLSO) approach is desirable for blockchains, the combination of these two properties introduces a new liveness concern. We show in our full paper [1] that faulty leaders can greatly reduce the throughput of *any* CLSO protocol. Worse, this attack does not require any explicit equivocation; it suffices for a faulty leader to delay responding, making it harder to detect. The root cause of the problem is simple: CLSO protocols require the formation of k QCs in *consecutive* views in order to commit a block (where $k \in \{2, 3\}$ depending on protocol). In the remainder of the paper, we refer to this constraint as the *consecutive honest leader condition (CHLC)*.

To the best of our knowledge, the aforementioned liveness concern is present in *all* CLSO protocols today, and thus represents a significant exploit opportunity for a Byzantine attacker. Even without malicious participants, expected network asynchrony can cause spurious view-changes, precluding the system from committing blocks. Consequently, this paper asks: is CHLC fundamental, or can it be relaxed?

2 Results

▶ **Definition 1** (Gap-Tolerance). *A BFT protocol achieves gap-tolerance if it requires k QCs in non-consecutive views to commit an operation.*

We observe that CLSO protocols *can* achieve gap-tolerance in some settings: commitment with non-consecutive QCs is possible when failures comprise of omission faults only. This observation can be coupled with the active detection of commission failures to determine when to (and when not to) relax the CHLC constraint. We prove the following theorem:

▶ **Theorem 2.** *There exists a CLSO protocol that is resilient to $f < n/3$ Byzantine replicas, is safe under asynchrony, and achieves gap-tolerance when failures are omission only.*

After GST and view synchronization, if views $v < v' < v''$ have honest leaders, and block B is proposed in view v, then either block B will be committed in view v'' or a previously undetected faulty replica will be detected and slashed (excluded from any future participation).

In this paper we propose Siesta (complete description is in our full paper [1]), a new consensus protocol that satisfies Theorem 2. Siesta either (i) allows blocks to commit in non-consecutive views or (ii) immediately and reliably detects equivocation and slashes (punishes and removes) a malicious leader.

Two fundamental ideas lie at the center of Siesta:

- *No-QC Proofs.* It is safe to commit a block in non-consecutive views as long as one can prove that no QC for a conflicting block could have formed in between views. Siesta carefully designs such proofs to be efficient.

- *Equivocation proofs.* Many BFT protocols do not include phase one messages in view changes as conflicting phase one messages can exist in the presence of malicious leaders. Although these messages cannot be used to successfully commit a block, we observe that they can be used as an explicit *proof of equivocation* to hold perpetrators accountable.

2.1 Performance Results and Complexity

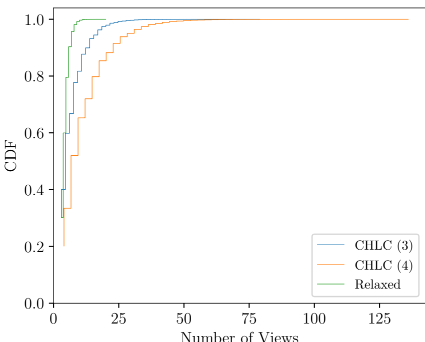


Figure 1 CDF of the number of views needed to commit an operation n = 100.

We quantify the performance gains made possible by relaxing the consecutive honest leader condition (CHLC). We compare Siesta to 1) two-phase CLSO protocols and 2) three-phase CLSO protocols. We assume a random leader election strategy; experiments with round-robin election convey similar results. We simulate each protocol and report how many rounds were necessary to satisfy each protocol's commit rule.

In CLSO protocols, the number of rounds directly influences both latency and throughput. If a round has latency x, then commit latency for an operation will be $x * rounds$ while throughput is calculated by dividing the batch size by the expected commit latency. We write CHLC(4) for protocols requiring four consecutive honest leaders, CHLC(3) for three consecutive honest leaders, and finally Relaxed for our own protocol Siesta. Figure 1 shows the resulting commit latency CDF. Siesta achieves an expected commit latency of 4.5 rounds; CHLC(3) requires $\approx$ 7 rounds. CHLC(4) has worst the expected performance, taking 12 rounds to commit. Worst-case commit latency is especially interesting: Siesta has relatively low worst-case latency, with 18 rounds, while CHLC(3) protocols have a worst-case commit time of 76 rounds. CHLC(4) has seven times worst latency, with a worst-case commit time of 129 rounds.

References

1 Ittai Abraham, Natacha Crooks, Neil Giridharan, Heidi Howard, and Florian Suri-Payer. It's not easy to relax: liveness in chained bft protocols, 2022. doi:10.48550/arXiv.2205.11652.
2 Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint*, 2017. doi:10.48550/arXiv.1710.09437.
3 Mohammad M. Jalalzai, Jianyu Niu, and Chen Feng. Fast-hotstuff: A fast and resilient hotstuff protocol, 2020. doi:10.48550/arXiv.2010.11454.
4 The Diem Team. DiemBFT v4: State machine replication in the diem blockchain, 2021.
5 Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, 2019.

Brief Announcement: Distributed Algorithms for Minimum Dominating Set Problem and Beyond, a New Approach

Sharareh Alipour[1]

Tehran Institute for Advanced Studies, Iran

Mohammadhadi Salari

Simon Fraser University, Burnaby, Canada

Abstract

In this paper, we study the minimum dominating set (MDS) problem and the minimum total dominating set (MTDS) problem. We propose a new idea to compute approximate MDS and MTDS. This new approach can be implemented in a distributed model or parallel model. We also show how to use this new approach in other related problems such as set cover problem and k-distance dominating set problem.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms; Theory of computation → Distributed algorithms

Keywords and phrases Minimum dominating set problem, set cover problem, k-distance dominating set problem, distributed algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.40

1 Introduction

Let $G = (V, E)$ be an undirected graph with the vertex set V and the edge set E and without isolated vertex. We denote the set of adjacent vertices to a vertex v, neighbors of v, by $N(v)$. A set $S \subseteq V$ is a dominating set of G if each node $v \in V$ is either in S or has a neighbor in S. Also, $S \subseteq V$ is a total dominating set of G if each node $v \in V$ has a neighbor in S. Let $\gamma(G)$ and $\gamma_t(G)$ be the size of a minimum dominating set (MDS) and a minimum total dominating set (MTDS) of G, respectively. It is easy to prove that

$$\gamma(G) \leq \gamma_t(G) \leq 2\gamma(G).$$

Also, a subset of vertices such that each edge of the graph G is incident to at least one vertex of the subset is a vertex cover. A minimum vertex cover (MVC) of G is a vertex cover with the smallest possible number of vertices. The size of MVC is denoted by $\beta(G)$. A subset of the vertices such that no two vertices in the subset represent an edge of G is an independent set of G. An independent set with the largest possible number of vertices is called a maximum independent set (MIS). The size of MIS is denoted by $\alpha(G)$.

An interesting problem is computing the MDS in the distributed model, which we will consider in this paper. In a distributed model the network is abstracted as a simple n-node undirected graph $G = (V, E)$. There is a processor on each node $v \in V$, with a unique $\Theta(\log n)$-bit identifier $ID(v)$, who initially knows only its neighbors in G. Communication happens in synchronous rounds. Per round, each node can send one, possibly different, $O(\log n)$-bit message to each of its neighbors. Ultimately, each node should know its own

[1] corresponding author

© Sharareh Alipour and Mohammadhadi Salari;
licensed under Creative Commons License CC-BY 4.0

part of the output. When computing the dominating set, each node knows whether it is in the dominating set or has a neighbor in the dominating set [2]. When the size of the messages is restricted to be $O(\log n)$, then the algorithm is CONGEST.

2 Theoretical result and the algorithm

For a given graph G, with no isolated vertex, we construct a graph G' with the same set of vertices as in G as follows. For each vertex v of degree at least two, we choose two of its neighbors arbitrarily and add an edge between them in G'. If v is of degree one, we add a loop edge on its neighbor. We call this edge the corresponding edge of v in G' and denote it by e_v. Note that if the graph G has a cycle of length 4, with vertices a, b, c, d then the edge bd can be the corresponding edge of both a and c in G'. Let $\alpha(G)$ and $\beta(G)$ be the size of a maximum independent set and the size of a minimum vertex cover of G, respectively (as defined previously). Then, we have the following theorem.

▶ **Theorem 1.** $\gamma_t(G) \leq n - \alpha(G') = \beta(G')$.

Proof. Suppose that D is a maximum independent set of G', so $|D| = \alpha(G')$. We show that $V \backslash D$ is a total dominating set for G. For each vertex v, we choose two of its neighbors, for example, u and w, and add an edge between them in G'. Since there is an edge between u and w, at most, one of them can be in D, which means at least one of them is in $V \backslash D$. The same argument applies when an edge is a loop. Thus, for each vertex v, at least one of its neighbors in G is in $V \backslash D$, so $V \backslash D$ is a total dominating set for G. The size of $|V \backslash D|$ equals $n - \alpha(G')$ and we have $\gamma_t(G) \leq n - \alpha(G') = \beta(G')$. ◀

Note that the graph G' can be constructed is a constant number of rounds in the CONGEST model. According to our time and space constraints, we can use the known distributed algorithms for computing a vertex cover for G' (See [3, 4]).

3 Extension to the other problems

Set cover problem

In the set cover problem we are given a set $A = \{a_1, a_2, \ldots a_n\}$ of n elements and m subsets, $A_1, A_2, \ldots, A_m$ of A. The goal is to choose the minimum number of subsets that cover all the elements of A.

Our algorithm is as follows. Each element a_i chooses a subset A_j with maximum size such that $a_i \in A_j$. Let x_i be the number of times that A_i is chosen by the elements of A. We construct a graph G' that its vertices are the subsets $A_1, A_2, \ldots, A_m$. For each $a \in A$ we choose two subsets A_i and A_j with maximum values of x_i's such that $a \in A_i$ and $a \in A_j$ and add an edge between them. Similar to the proof of Theorem 1, it can be shown that a vertex cover for G' is a set cover for A.

k-distance dominating set

An extension of the MDS problem is the minimum k-distance dominating set problem where the goal is to choose a subset $S \subseteq V$ with the minimum cardinality such that for every vertex $v \in V \backslash S$, there is a vertex $u \in S$ where the shortest path between them at most k. The minimum total k-distance dominating set is defined similarly. A k-observer Ob of a network N is a set of nodes in N such that each message, that travels at least k hops in N, is handled (and so observed) by at least one node in Ob. A k-observer Ob of a network N is minimum

iff the number of nodes in Ob is less than or equal to the number of nodes in every k-observer of N (See [1]). This problem is equivalent to the k-distance dominating set problem. In this problem for each node v, the neighbors of v, is the set of nodes whose distance from v is less than $k+1$. Then we apply the proposed algorithms as before.

Note that computing a minimum k-distance dominating set for a graph G is equivalent to computing a minimum dominating set for G^k, where G^k is a graph with the same vertex set as G and we put an edge between two vertices in G^k if the distance between them in G is less than $k+1$.

References

1 Krishnendu Chakrabarty, S. Sitharama Iyengar, Hairong Qi, and Eungchun Cho. Grid coverage for surveillance and target location in distributed sensor networks. *IEEE Trans. Computers*, 51(12):1448–1453, 2002. `doi:10.1109/TC.2002.1146711`.

2 Mohsen Ghaffari and Fabian Kuhn. Derandomizing distributed algorithms with small messages: Spanners and dominating set. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, pages 29:1–29:17, 2018. `doi:10.4230/LIPIcs.DISC.2018.29`.

3 Fabrizio Grandoni, Jochen Könemann, and Alessandro Panconesi. Distributed weighted vertex cover via maximal matchings. *ACM Trans. Algorithms*, 5(1):6:1–6:12, 2008. `doi:10.1145/1435375.1435381`.

4 Michal Hanckowiak, Michal Karonski, and Alessandro Panconesi. On the distributed complexity of computing maximal matchings. *SIAM J. Discret. Math.*, 15(1):41–57, 2001. `doi:10.1137/S0895480100373121`.

Brief Announcement: Survey of Persistent Memory Correctness Conditions

Naama Ben-David ✉
VMware Research, Palo Alto, CA, USA

Michal Friedman ✉
ETH Zürich, Switzerland

Yuanhao Wei ✉
Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

In this brief paper, we survey existing correctness definitions for concurrent persistent programs.

2012 ACM Subject Classification Hardware → Memory and dense storage; Theory of computation

Keywords and phrases Persistence, NVRAM, Correctness, Concurrency

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.41

Related Version *Full Version*: https://arxiv.org/pdf/2208.11114.pdf

1 Introduction

Non-Volatile Random Access Memory (NVRAM) is a new type of memory technology that has recently hit the market. Its key feature is that it is *persistent*, like SSDs, but is fast and byte-addressable, much like DRAM. This presents a huge paradigm shift from the way persistence could be achieved in the past; techniques that worked well for sequential block-granularity storage cannot be efficiently used with NVRAMs. Achieving persistence with NVRAM has the potential to speed up applications by orders of magnitude.

However, before designing persistent algorithms for NVRAM, we must first answer a more basic question: What does it mean for an algorithm to be persistent?

Despite algorithms relying on external storage for persistence for decades, the answer to the above question is not clear in the context of faster, byte addressible NVRAM. In particular, it is now realistic to require that virtually *no progress be lost* upon a crash, and that a program be able to continue where it left off upon recovery.

The above requirement, while appealing, is in fact not very precise. Due to registers and caches remaining volatile, individual instructions and memory accesses are applied to volatile memory first, and are then persisted separately. If a system crash occurs between when an instruction is executed and when its effect is applied to NVRAM, progress will inevitably be lost. However, it is possible to define how much progress it is okay to lose, and at what point in the execution we expect each instruction's effect to be persisted. For example, we can ensure no completed operation will be lost upon a crash.

In this brief survey, we discuss definitions of persistence that exist in the literature. As this is an actively and quickly developing field of study, there are many different notations, terminologies, and definitions that often refer to similar notions. We put these definitions into the same terminology, and compare them to each other. Using this point of view, we arrange the definitions into a hierarchy, based on the set of execution histories that satisfies every definition. Interestingly, this hierarchy changes depending on specific model assumptions made. We outline common model assumptions and illustrate their effect on these definitions.

© Naama Ben-David, Michal Friedman, and Yuanhao Wei;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 41; pp. 41:1–41:4
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We note that this survey is meant to make sense of the various persistence definitions and to guide researchers and algorithm designers when choosing which model and definition to adopt. However, this survey does not cover the many different algorithms, techniques, and applications that have been developed for NVRAM programming in recent years.

2 Model Assumptions

We consider a system of n asynchronous processes $p_1 \ldots p_n$. Processes may access *shared base objects* with atomic *read, write, and read-modify-write* primitives. Each process also has access to *local variables* that are not shared with any other process. Objects (both local and shared) may be *volatile* or *non-volatile*, which affects their behavior upon a crash.

A *history* is a sequence of *events*. There are three types of events: an *invocation* event $obj.inv_i(op, v)$ which invokes operation *op* on object *obj* by process p_i with argument v, a *response* event $obj.res_i(op, v)$ in which *obj* responds to p_i's invocation on *op* with return value v, and *crash* events. A crash resets all the volatile local variables of the associated process, or all volatile objects if all processes crashed.

A response *res* is said to *match* an invocation *inv* in H if *obj*, *op*, and i are the same for both, and *res* is the next event in $H|p$ after *inv*. An operation is said to be *complete* in H if both its invocation and a matching response appear in H. Otherwise, if an operation was invoked but was not completed, the operation is said to be *pending*.

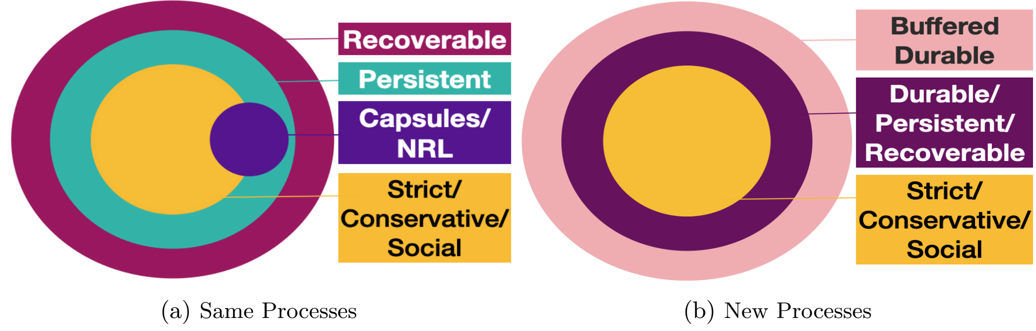
Given a property P and a history H, P is said to be *local* if given a history H in which, for every object O, $H|O$ satisfies P, H also satisfies P.

In the full version of this paper [4], we discuss model variants that appear in the literature and their implications on correctness conditions and implementations.

3 Property Hierarchy

In this section, we present hierarchies relating the existing properties to each other under various model assumptions. A complete list of all formal definitions, including those omitted from this short paper [2, 3, 6, 7, 10], and a more profound comparison among them can be found in the full version [4].

3.1 Same Processes are Invoked



(a) Same Processes (b) New Processes

Figure 1 Hierarchy of definitions when the same processes and new processes are allowed to be invoked after a crash.

In this subsection, we assume that the model allows the same processes to be invoked after a crash. Under this model, the existing definitions can be arranged into the hierarchy that is presented in Figure 1(a). The hierarchy is based on the sets of execution histories that are allowed by each of the definitions; in Figure 1(a), each definition's set of allowed histories is represented by its labelled region.

To understand this hierarchy, it is useful to consider how each correctness condition allows linearizing a given history. The correctness conditions differ by where they allow each pending operation's completion to be placed. Berryhill et al. [5] presented recoverable, persistent, and strict linearizability in this light.

There are several points in a given history with respect to which it may make sense to complete such a pending operation. One point of reference is the crash event that immediately follows inv_{op} in $H|p$. Another is the next invocation by p in the history. Finally, we may also consider the next invocation in H that occurs in the same object as op.

Strict linearizability [1] is the strongest (or *strictest*) condition, in that it allows for the smallest set of histories. It requires every pending operation to be eliminated or completed before the crash. In addition, it is local, meaning that every object that is built from strictly linearizable objects is also strictly linearizable. To achieve this guarantee, one may think of running a recovery operation directly after the crash, and before executing the program. However, it might be too restrictive; in some scenarios, it makes sense to relax this requirement to allow recovery (alternatively; the completion of pending operations) to occur later in the execution.

While strict linearizability requires completions to be placed before the next crash event, persistent atomicity [8] instead completes operations before the next invocation by the same process. Note that, by the definition of legal histories, the next invocation by the same process can never be placed before the current crash, and therefore persistent atomicity is weaker than strict linearizability (i.e. the set of persistent histories *contains* the set of strict histories). Due to this relaxation, it is not local. On the positive side, persistent atomicity may be easier to implement than strict linearizability, since an operation only needs to be recovered (if ever) when the same process invokes another operation.

Berryhill et al. [5] presented the recoverable linearizability definition, which is the most relaxed one. It also requires pending operations to be completed (or removed) before the crash, but practically, to "take effect" before the next invocation of the same process on the same object. Therefore, it allows the most extensive set of histories.

3.2 New processes are invoked

In this subsection, we assume that the model does not allow the same processes to be invoked after a crash, and new processes are spawned instead. This model was first suggested by Izraelevitz et al. [9], as a simplification to previous models. Under this simplification, the definitions that deal with execution continuations do not make sense. The hierarchy in this model is presented in Figure 1(b).

When the same processes are never invoked after a crash, strict linearizability [1] still remains the strongest condition as it requires every pending operation to be eliminated or completed before the crash. Recall that the difference between persistent atomicity [8] and recoverable linearizability [5] is only in recoveries by the same process, and thus these two definitions have the same meaning as durable linearizability [9] which requires getting a linearizable history after removing all crash events from the original history.

By disallowing the executions of the same processes, durable linearizability, persistent atomicity and recoverable linearizability are local under this restriction. Buffered durable linearizability [9] is similar to the others, but additionally allows operations that were completed before the crash to be removed. It therefore is the weakest definition.

References

1 Marcos K Aguilera and Svend Frølund. Strict linearizability and the power of aborting. *Technical Report HPL-2003-241*, 2003.

2 Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 7–16. ACM, 2018.

3 Naama Ben-David, Guy E Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–264. ACM, 2019.

4 Naama Ben-David, Michal Friedman, and Yuanhao Wei. Survey of persistent memory correctness conditions. *arXiv preprint*, 2022. `arXiv:2208.11114`.

5 Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

6 Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *ACM SIGPLAN Notices*, pages 28–40. ACM, 2018.

7 Wojciech Golab and Aditya Ramaraju. Recoverable mutual exclusion. *Distributed Computing*, 32(6):535–564, 2019.

8 Rachid Guerraoui and Ron R Levy. Robust emulations of shared memory in a crash-recovery model. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, pages 400–407. IEEE, 2004.

9 Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing*, pages 313–327. Springer, 2016.

10 Nan Li and Wojciech Golab. Detectable sequential specifications for recoverable shared objects. In *35th International Symposium on Distributed Computing (DISC 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

Brief Announcement: Minimizing Congestion in Hybrid Demand-Aware Network Topologies

Wenkai Dai
Faculty of Computer Science, Universität Wien, Austria

Michael Dinitz
Computer Science Department, Johns Hopkins University, Baltimore, MD, USA

Klaus-Tycho Foerster
Computer Science Department, Technische Universität Dortmund, Germany

Stefan Schmid
TU Berlin, Germany
Faculty of Computer Science, Universität Wien, Austria

Abstract

Emerging reconfigurable optical communication technologies enable demand-aware networks: networks whose static topology can be enhanced with demand-aware links optimized towards the traffic pattern the network serves. This paper studies the algorithmic problem of how to jointly optimize the topology and the routing in such demand-aware networks, to minimize congestion. We investigate this problem along two dimensions: (1) whether flows are splittable or unsplittable, and (2) whether routing on the hybrid topology is segregated or not, i.e., whether or not flows either have to use exclusively either the static network or the demand-aware connections. For splittable and segregated routing, we show that the problem is 2-approximable in general, but APX-hard even for uniform demands induced by a bipartite demand graph. For unsplittable and segregated routing, we show an upper bound of $O(\log m / \log\log m)$ and a lower bound of $\Omega(\log m / \log\log m)$ for polynomial-time approximation algorithms, where m is the number of static links. Under splittable (resp., unsplittable) and non-segregated routing, even for demands of a single source (resp., destination), the problem cannot be approximated better than $\Omega(c_{\max}/c_{\min})$ unless P=NP, where $c_{\max}$ (resp., $c_{\min}$) denotes the maximum (resp., minimum) capacity. It is still NP-hard for uniform capacities, but can be solved efficiently for a single commodity and uniform capacities.

2012 ACM Subject Classification Networks → Network architectures; Theory of computation → Design and analysis of algorithms

Keywords and phrases Congestion, Reconfigurable Networks, Algorithms, Complexity

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.42

Funding Research supported by the European Research Council (ERC), grant agreement No. 864228 (AdjustNet) Horizon 2020, 2020-2025, and NSF Award CCF-1909111.

1 Introduction

Emerging demand-aware networks, whose topologies are typically hybrid, in that a static (and demand-oblivious) network is enhanced with reconfigurable (and demand-aware) links, introduce unprecedented flexibility in adapting the network topology towards current traffic demands. In such hybrid networks, the reconfigurable links are usually enabled by optical circuit switches [1, 8, 13], and particularly, each optical circuit switch provides reconfigurable links by establishing connections between pairs of its ports, i.e., a matching.

Extensive past works studied the question of how to jointly optimize topology and routing of such hybrid (reconfigurable) networks [17] for different networking performance metrics, e.g., latency [11], throughput [4, 7], routing length [14, 15, 16], flow times [3] etc. Interestingly, *min-congestion*, a most central performance metric in traditional networks, is still not well-understood in hybrid networks. Avin et al. [6] and Pacut et al. [12] study optimal

© Wenkai Dai, Michael Dinitz, Klaus-Tycho Foerster, and Stefan Schmid;
licensed under Creative Commons License CC-BY 4.0

Table 1 Summary of our approximation upper and lower bounds on the MCHN problem (Def. 1).

Approximation Upper & Lower Bounds (Complexity)	Splittable Flow	Segregated Routing	Restrictions On Demands
2-approximation	yes	yes	
APX-complete	yes	yes	uniform and bipartite demands
$O(\log m/\log\log m)$-approximation	no	yes	
Lower Bound: $\Omega(\log m/\log\log m)$	no	both	
$(2 \cdot c_{\max}/c_{\min})$-approximation	yes	no	single source (resp., dest.)
Lower Bound: $\Omega(c_{\max}/c_{\min})$	both	no	single source (resp., dest.)

bounded-degree topology designs to minimize both the route length and the congestion. Dai et al. [2] worked on the same network model as us, showing that the problem is already NP-hard for *splittable* (resp., *unsplittable*) and *segregated* (resp., *non-segregated*) routing models when the static network is a tree of height at least two, but tractable for static networks of star topologies. Zheng et al. [10] introduced a greedy-based heuristic algorithm for our *segregated* model but on specific topologies of datacenters. However, not much more is known w.r.t. corresponding approximation bounds, which motivates our study, summarized in Table 1.

2 Model

Network Model. We consider a *hybrid* network [5, 9] $N = (V, E, \mathcal{E}, c)$, where a *static* network (V, E) is represented by an bidirected (simple) graph of nodes V, any two distinct nodes $v_i, v_j \in V$ imply a possible *reconfigurable link* denoted by a bidirected edge $\{i, j\}$ in $\mathcal{E}$, and a function $c : \vec{E} \cup \vec{\mathcal{E}} \mapsto \mathbb{R}_{\geq 0}$ defines *capacities* for both directions of each bidirected link in $E \cup \mathcal{E}$ with the *maximum* (resp., *minimum*) *capacity* denoted by $c_{\max}$ (resp., $c_{\min}$). The hybrid network N must decide a matching $M \subseteq \mathcal{E}$ to obtain an enhanced graph $N(M) = (V, E \cup M, c)$, which determines the actual topology of the communicating network.

Traffic Demands. A certain communication pattern (*demands*) on nodes V is represented by a matrix $D := (d_{i,j})_{|V|\times|V|}$, where an entry $d_{i,j} \in \mathbb{R}_{\geq 0}$ denotes the traffic load (frequency) or a demand from the node $v_i \in V$ to the node $v_j \in V$.

Routing Models. The *unsplittable* routing requires that flows of each demand must be sent along a single (directed) path, otherwise the routing is called *splittable*. For a hybrid network, *segregated* routing requires that each demand $d_{i,j}$ is either sent on the reconfigurable link $\{i, j\}$, if it exists, or purely on the static network, otherwise it is *unsegregated* routing. Hence, we consider *four different routing models*: *Unsplittable & Segregated (US)*, *Unsplittable & Non-segregated (UN)*, *Splittable & Segregated (SS)*, and *Splittable & Non-segregated (SN)*.

▶ **Definition 1** (Min-Congestion Hybrid Network Problem (MCHN)). *Given a hybrid network $N = (V, E, \mathcal{E}, c)$, a routing model $\tau \in \{US, UN, SS, SN\}$, and a demand matrix D, find a matching $M \subseteq \mathcal{E}$, s.t., the congestion λ, i.e., the maximum load on $\vec{E} \cup \vec{M}$, to serve D in $N(M)$ is minimized.*

3 Our Contributions

We initiate the study of approximation algorithms for minimizing congestion in hybrid demand-aware networks (for a given matrix of demands). Our results include an overview of approximation results and complexity characterizations in general settings. We also provide a fine-grained algorithmic analysis for restricted cases:

Segregated Routing. We can give a mixed-integer programming formulation for segregated and un-/splittable flow models whose LP relaxation can be solved efficiently. For splittable flows, we provide a 2-approximation algorithm by a novel deterministic rounding approach, and also prove APX-hardness even if demands are uniform and bipartite. However, we also show that the problem becomes tractable for demands with a single source (resp., dest.). For unsplittable flows, we show that the hybrid network problem cannot be approximated better than the min-congestion multi-commodity unsplittable flow problem (MCMF) [18], but any ρ-approximation algorithm based on rounding techniques for the MCMF problem can be utilized to give a 2ρ-approximation. This implies an approximability of $\Theta(\log m/\log\log m)$ for segregated and unsplittable routing, where $m = |E|$.

Non-Segregated Routing. Under the splittable (resp., unsplittable) flow model, even for demands of a single source (resp., destination), the problem cannot be approximated better than $\Omega(c_{\max}/c_{\min})$ unless P=NP, but still $(2 \cdot c_{\max}/c_{\min})$-approximable for the splittable flow, where $c_{\max}$ (resp., $c_{\min}$) denotes the maximum (resp., minimum) capacity on all links, and it still remains NP-hard for *uniform capacities*, i.e., $c : \vec{E} \cup \vec{\mathcal{E}} \mapsto \{a\}$ for $a \in \mathbb{R}_{>0}$. However, the problem with uniform capacities becomes efficiently solvable for demands of a single commodity under un-/splittable flow.

References

1. S. Aleksic. The future of optical interconnects for data centers: A review of technology trends. In *2017 14th International Conference on Telecommunications (ConTEL)*, June 2017.
2. W. Dai et al. Load-optimization in reconfigurable networks: Algorithms and complexity of flow routing. *SIGMETRICS Perform. Evaluation Rev.*, 48(3), 2020.
3. M. Dinitz and B. Moseley. Scheduling for weighted flow and completion times in reconfigurable networks. In *INFOCOM*. IEEE, 2020.
4. A. Singla et al. High throughput data center topology design. In *NSDI*. USENIX, 2014.
5. B. Venkatakrishnan et al. Costly circuits, submodular schedules and approximate carathéodory theorems. In *SIGMETRICS*, 2016.
6. C. Avin et al. Demand-aware network design with minimal congestion and route lengths. In *INFOCOM*. IEEE, 2019.
7. D. Nikhil et al. Stable matching algorithm for an agile reconfigurable data center interconnect (MSR-TR-2016-1140). Technical report, Microsoft Research, June 2016.
8. G. Wang et al. c-through: part-time optics in data centers. In *SIGCOMM*. ACM, 2010.
9. H. Liu et al. Scheduling techniques for hybrid circuit/packet networks. In *CoNEXT*. ACM, 2015.
10. J. Zheng et al. Dynamic load balancing in hybrid switching data center networks with converters. In *ICPP*. ACM, 2019.
11. M. Ghobadi et al. Projector: Agile reconfigurable data center interconnect. In *SIGCOMM*. ACM, 2016.
12. M. Pacut et al. Improved scalability of demand-aware datacenter topologies with minimal route lengths and congestion. *Perform. Evaluation*, 152, 2021.
13. N. Farrington et al. Helios: a hybrid electrical/optical switch architecture for modular data centers. In *SIGCOMM*. ACM, 2010.
14. T. Fenz et al. Efficient non-segregated routing for reconfigurable demand-aware networks. *Comput. Commun.*, 164, 2020.
15. K.-T. Foerster et al. On the complexity of non-segregated routing in reconfigurable data center architectures. *Comput. Commun. Rev.*, 49(2):2–8, 2019.
16. K.-T. Foerster and S. Schmid. Survey of reconfigurable data center networks: Enablers, algorithms, complexity. *SIGACT News*, 50(2), 2019.
17. M. N. Hall et al. A survey of reconfigurable optical networks. *Opt. Switch. Netw.*, 41, 2021.
18. Vijay V. Vazirani. *Approximation algorithms*. Springer, 2001. URL: `http://www.springer.com/computer/theoretical+computer+science/book/978-3-540-65367-7`.

Brief Announcement: Computing Power of Hybrid Models in Synchronous Networks

Pierre Fraigniaud
IRIF, Université Paris Cité and CNRS, France

Pedro Montealegre[1]
Faculty of Engineering and Science, Universidad Adolfo Ibáñez, Santiago, Chile

Pablo Paredes
Department of Mathematical Engineering, University of Chile, Santiago, Chile

Ivan Rapaport
DIM-CMM (UMI 2807 CNRS), University of Chile, Santiago, Chile

Martín Ríos-Wilson
Faculty of Engineering and Science, Universidad Adolfo Ibáñez, Santiago, Chile

Ioan Todinca
LIFO, Université d'Orléans, France
INSA Centre-Val de Loire, Bourges, France

Abstract

During the last two decades, a small set of distributed computing models for networks have emerged, among which LOCAL, CONGEST, and Broadcast Congested Clique (BCC) play a prominent role. We consider *hybrid* models resulting from combining these three models. That is, we analyze the computing power of models allowing to, say, perform a constant number of rounds of CONGEST, then a constant number of rounds of LOCAL, then a constant number of rounds of BCC, possibly repeating this figure a constant number of times. We specifically focus on 2-round models, and we establish the complete picture of the relative powers of these models. That is, for every pair of such models, we determine whether one is (strictly) stronger than the other, or whether the two models are incomparable.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases hybrid model, synchronous networks, LOCAL, CONGEST, Broadcast Congested Clique

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.43

Related Version *Full Version*: https://doi.org/10.48550/arXiv.2208.02640

Funding Additional support for ANR projects QuData and DUCAT. This work was supported by Centro de Modelamiento Matemático (CMM), ACE210010 and FB210005, BASAL funds for centers of excellence from ANID-Chile, FONDECYT 11190482, FONDECYT 1220142 and PAI77170068.

1 Introduction

This paper analyzes the relative power of distributed computing models for networks, all resulting from the combination of standard synchronous models such as LOCAL and CONGEST [4], as well as Broadcast Congested Clique (BCC) [1]. Each of these three models has its strengths and limitations. We investigate the power of models resulting from combining these three models, in order to take advantage of their positive aspects without suffering from their negative ones.

[1] Corresponding Author

© Pierre Fraigniaud, Pedro Montealegre, Pablo Paredes, Ivan Rapaport, Martín Ríos-Wilson, and Ioan Todinca;
licensed under Creative Commons License CC-BY 4.0

For the sake of comparing models, we focus on the standard framework of distributed *decision* problems on labeled graphs (see [2]). Such problems are defined by a collection $\mathcal{L}$ of pairs (G, ℓ), where $G = (V, E)$ is a graph, and $\ell : V \to \{0,1\}^*$ is a function assigning a label $\ell(u) \in \{0,1\}^*$ to every $u \in V$. Such a set $\mathcal{L}$ is called a distributed *language*. A distributed algorithm A *decides* $\mathcal{L}$ if every node running A eventually accepts or rejects, and the following condition is satisfied: for every labeled graph (G, ℓ), every node should accept in a yes-instance (i.e., an instance $(G, \ell) \in \mathcal{L}$), and, in a no-instance (i.e., an instance $(G, \ell) \notin \mathcal{L}$), at least one node must reject.

For every $t \geq 0$, let us denote by $\mathbf{L}^t$ the set of distributed languages $\mathcal{L}$ for which there is a t-round algorithm in the LOCAL model deciding $\mathcal{L}$. The sets $\mathbf{C}^t$ and $\mathbf{B}^t$ are defined similarly, for the CONGEST and BCC models, respectively. Note that while it is easy to show, using indistinguishability arguments, that, for every $t \geq 1$, $\mathbf{L}^t \smallsetminus \mathbf{L}^{t-1} \neq \varnothing$ and $\mathbf{C}^t \smallsetminus \mathbf{C}^{t-1} \neq \varnothing$, establishing that there is indeed a decision problem in $\mathbf{B}^t \smallsetminus \mathbf{B}^{t-1}$ requires significantly more work [3]. Also, we define $\mathbf{L}^* = \cup_{t \geq 0} \mathbf{L}^t$, $\mathbf{C}^* = \cup_{t \geq 0} \mathbf{C}^t$, and $\mathbf{B}^* = \cup_{t \geq 0} \mathbf{B}^t$. So, in particular, $\mathbf{L}^*$ is the class of distributed languages that can be decided in a constant number of rounds in the LOCAL model.

2 Our Results

On the negative side, we provide a series of separation results between 2-round hybrid models. In particular, we show that **BC** and **CB** are incomparable. That is, there are languages in $\mathbf{BC} \smallsetminus \mathbf{CB}$, and languages in $\mathbf{CB} \smallsetminus \mathbf{BC}$. In fact, we show stronger separation results, by establishing that $\mathbf{BC} \smallsetminus \mathbf{C}^*\mathbf{B} \neq \varnothing$, and $\mathbf{CB} \smallsetminus \mathbf{BL}^* \neq \varnothing$. That is, in particular, there are languages that can be decided by a 2-round algorithm performing a single BCC round followed by one CONGEST round, which cannot be decided by any algorithm performing k CONGEST rounds followed by a single BCC round, for any $k \geq 1$.

On the positive side, we show that, for any non-negative integers $\alpha_1, \dots, \alpha_k, \beta_1, \dots, \beta_k$,

$$\prod_{i=1}^{k} \mathbf{L}^{\alpha_i} \mathbf{B}^{\beta_i} \subseteq \mathbf{L}^{\sum_{i=1}^{k} \alpha_i} \mathbf{B}^{\sum_{i=1}^{k} \beta_i}. \tag{1}$$

That is, if a language $\mathcal{L}$ can be decided by a t-round algorithm alternating LOCAL and BCC rounds, then $\mathcal{L}$ can be decided by a t-round algorithm performing all its LOCAL rounds first, and then all its BCC rounds – with the notations of Eq. (1), $t = \sum_{i=1}^{k} (\alpha_i + \beta_i)$. So, in particular $\mathbf{BL} \subseteq \mathbf{LB}$. This inclusion is strict, since, as said before, $\mathbf{CB} \smallsetminus \mathbf{BL}^* \neq \varnothing$. In fact, this separation holds even if the number of LOCAL rounds depends on the number of nodes n in the network, as long as the algorithm performs $o(n)$ LOCAL rounds after its BCC round. Another consequence of Eq. (1) is that the largest class of languages among all the ones considered in this paper is $\mathbf{L}^*\mathbf{B}^*$, that is, languages that can be decided by algorithms performing k LOCAL rounds followed by k' BCC rounds, for some $k \geq 0$ and $k' \geq 0$.

Interestingly, our separation results hold even for randomized protocols, which can err with probability at most $\epsilon \leq 1/5$. That is, in particular, there is a language $\mathcal{L} \in \mathbf{CB}$ (i.e., that can be decided by a deterministic 2-round algorithm) which cannot be decided with error probability at most $1/5$ by any randomized algorithm performing one BCC round first, followed by k LOCAL rounds, for any $k \geq 1$. All our results about 2-rounds hybrid models are summarized on Figure 1.

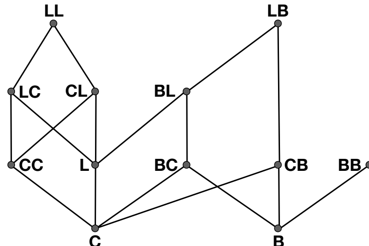


Figure 1 *The poset of 2-round hybrid models. An edge between a set of languages* $\mathbf{S}_1$ *and a set* $\mathbf{S}_2$*, where* $\mathbf{S}_1$ *is at a level lower than* $\mathbf{S}_2$*, indicates that* $\mathbf{S}_1 \subseteq \mathbf{S}_2$*. In fact, all inclusions are strict. Transitive edges are not displayed. Two sets that are not connected by a monotone path are incomparable.*

3 Our Techniques

All our separation results are obtained by reductions from communication complexity lower bounds. However, we had to revisit several known communication complexity results for adapting them to the setting of distributed decision, in which no-instances may be rejected by a single node, and non necessarily by all the nodes. In particular, we revisit the classical `Index` problem. Recall that, in this problem, Alice is given a binary vector $x \in \{0,1\}^n$, Bob is given an index $i \in [n]$, and Bob must output x_i based on a single message received from Alice (1-way communication). We define the `XOR-Index` problem, in which Alice is given a binary vector $x \in \{0,1\}^n$ together with an index $i \in [n]$, Bob is given a binary vector $y \in \{0,1\}^n$ together with an index $j \in [n]$, and, after a single round of 2-way communication, Alice must output a boolean out_A and Bob must output a boolean out_B, such that $\text{out}_A \wedge \text{out}_B = x_j \oplus y_i$.

That is, if $x_j \neq y_i$ then Alice and Bob must both accept (i.e., output *true*), and if $x_j = y_i$ then *at least* one of these two players must reject (i.e., output *false*). We show that the sum of the sizes of the message sent by Alice to Bob and the message sent by Bob to Alice is $\Omega(n)$ bits. This bound holds even if the communication protocol is randomized and may err with probability at most $1/5$, and even if the two players have access to shared random coins.

The fact that only one of the two players may reject a no-instance (i.e., an instance where $x_j \oplus y_i = 0$), and not necessarily both, while a yes-instance must be accepted by both players, yields an asymmetry which complicates the analysis. We use information theoretic tools for establishing our lower bound. Specifically, we identify a way to decorrelate the behaviors of Alice and Bob, so that to analyze separately the distribution of decisions taken by each player, and then to recombine them for lower bounding the probability of error in case the messages exchanged between the players are small, contradicting the fact that this error probability is supposed to be small.

References

1. Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, pages 367–376, 2014.
2. Laurent Feuilloley and Pierre Fraigniaud. Survey of distributed decision. *Bull. EATCS*, 119, 2016.
3. Noam Nisan and Avi Widgerson. Rounds in communication complexity revisited. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 419–429, 1991.
4. David Peleg. *Distributed Computing: A Locality-Sensitive Approach.* SIAM, 2000.

Brief Announcement: New Clocks, Fast Line Formation and Self-Replication Population Protocols

Leszek Gąsieniec
University of Liverpool, UK

Paul Spirakis
University of Liverpool, UK

Grzegorz Stachowiak
Uniwersytet Wrocławski, Poland

Abstract

In this paper we consider a known variant of the standard population protocol model in which agents can be connected by edges, referred to as the *network constructor* model. During an interaction between two agents the relevant connecting edge can be formed, maintained or eliminated by the transition function. The state space of agents is fixed (constant size) and the size n of the population is not known, i.e., not hard-coded in the transition function.

Since pairs of agents are chosen uniformly at random the status of each edge is updated every $\Theta(n^2)$ interactions in expectation which coincides with $\Theta(n)$ parallel time. This phenomenon provides a natural lower bound on the time complexity for any non-trivial network construction designed for this variant. This is in contrast with the standard population protocol model in which efficient protocols operate in $O(\text{poly} \log n)$ parallel time.

The main focus in this paper is on efficient manipulation of linear structures including formation, self-replication and distribution (including pipelining) of *complex information* in the adopted model.

- We propose and analyse a novel edge based phase clock counting parallel time $\Theta(n \log n)$ in the network constructor model, showing also that its leader based counterpart provides the same time guaranties in the standard population protocol model. Note that all currently known phase clocks can count parallel time not exceeding $O(\text{poly} \log n)$.
- The new clock enables a nearly optimal $O(n \log n)$ parallel time spanning line construction (a key component of universal network construction), which improves dramatically on the best currently known $O(n^2)$ parallel time protocol, solving the main open problem in the considered model [9].
- We propose a new *probabilistic bubble-sort* algorithm in which random comparisons and transfers are allowed only between the adjacent positions in the sequence. Utilising a novel potential function reasoning we show that rather surprisingly this probabilistic sorting (via conditional pipelining) procedure requires $O(n^2)$ comparisons in expectation and whp, and is on par with its deterministic counterpart.
- We propose the first population protocol allowing self-replication of a *strand* of an arbitrary length k (carrying a k-bit message of size independent of the state space) in parallel time $O(n(k + \log n))$. The pipelining mechanism and the time complexity analysis of the strand self-replication protocol mimic those used in the probabilistic bubble-sort. The new protocol permits also simultaneous self-replication, where l copies of the strand can be created in time $O(n(k + \log n) \log l)$. Finally, we discuss application of the strand self-replication protocol to pattern matching.

Our protocols are always correct and provide time guaranties with high probability defined as $1 - n^{-\eta}$, for a constant $\eta > 0$.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Population protocols, network constructors, probabilistic bubble-sort, self-replication

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.44

Related Version *Full Version*: https://arxiv.org/abs/2111.10822

© Leszek Gąsieniec, Paul Spirakis, and Grzegorz Stachowiak;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

The model of *population protocols* originates from the seminal paper of Angluin et al. [1]. This model provides tools for the formal analysis of *pairwise interactions* between simple indistinguishable entities referred to as *agents*. The agents are equipped with limited storage, communication and computation capabilities. When two agents engage in a direct interaction their states are amended according to the predefined *transition function*. The weakest possible assumptions in population protocols, also adopted here, limit the state space of agents to a fixed (constant) size disallowing utilisation of the size of the population n in the transition function. In the *probabilistic variant* of population protocols adopted in this paper, in each step the *random scheduler* selects from the whole population an ordered pair of agents formed of the *initiator* and the *responder*, uniformly at random. The lack of symmetry in this pair is a powerful source of random bits often used by population protocols. In this variant, in addition to *state utilisation* one is also interested in the *time complexity* of the proposed solutions. In more recent work on population protocols the focus is on *parallel time* defined as the total number of pairwise interactions (sequential time) leading to the solution divided by the size n of the whole population. For example, a core dissemination tool in population protocols known as *one-way epidemic* [2] distributes simple (e.g., 0/1) messages to all agents in the population utilising $\Theta(n \log n)$ interactions or equivalently $\Theta(\log n)$ parallel time. The parallel time is meant to reflect on massive parallelism of simultaneous interactions. While this is a simplification [4], it provides a good estimation on locally observed time expressed in the number of interactions each agent was involved in throughout the computation process.

Unless stated otherwise we assume that any protocol starts in the predefined *initial configuration* with all agents being in the same *initial state*. A population protocol *terminates with success* if the whole population stabilises eventually, i.e., it arrives at and stays indefinitely in the *final configuration* of states representing the desired property of the solution.

1.1 Our results and their significance

We study here several central problems in distributed computing by focusing on the adopted variant of population protocols. These include the concept of *phase clocks*, a distributed synchronisation tool with good space and accuracy guarantees. The first study of leader based $O(1)$ space phase clocks can be found in the seminal paper by Angluin *et al.* in [2]. Further extensions including the junta based clock and nested clocks counting any $\Theta(\text{poly} \log n)$ parallel time were analysed in [6]. In a very recent work [5] Doty *et al.* study constant resolution phase clocks utilising $O(\log n)$ states as the main engine in the optimal *majority* computation protocols. In this work we propose and analyse a new phase clock based on a matching allowing to count $\Theta(n \log n)$ parallel time. This is the first clock able to confirm the conclusion of the slow leader election protocol based on direct duels between the (remaining) leader candidates. We also propose an edge-less variant of this clock based on the computed leader. This clock powers a nearly optimal $O(n \log n)$ parallel time spanning line construction (a key component of universal network construction), improving dramatically on the best currently known $O(n^2)$ parallel time protocol, solving the main open problem from [9].

We also consider a probabilistic variant of the classical bubble-sort algorithm, in which any two consecutive positions in the sequence are chosen for comparison uniformly at random. We show that rather surprisingly this variant is on par with its deterministic counterpart as it requires $\Theta(n^2)$ random comparisons whp. While this new result is of an independent algorithmic interest, together with the edge-less clock they conceptually power the strand (line-segment carrying information) self-replication protocol studied at the end of this paper.

In a wider context, *self-replication* is a property of a dynamical system which allows reproduction. Such systems are of increasing interest in biology, e.g., in the context of how life could have begun on Earth [8], but also in computational chemistry [10], robotics [7] and other fields. In our case a larger chunk of information (well beyond the limited state capacity) is stored collectively in a strand (line-segment) of agents. Such strands may represent strings in pattern matching or a large value, or a code in more complex distributed process. In such cases the replication mechanism facilitates an improved accessibility to this information. We propose the first strand self-replication protocol allowing to reproduce a strand of size k in parallel time $O(n(k + \log n))$. This protocol permits simultaneous replication, where l copies of a strand can be generated in parallel time $O(n(k + \log n) \log l)$. The parallelism of this protocol is utilised in efficient pattern matching.

The full version of this paper including definitions, algorithms and formal proofs is available on arXiv.org [3].

2 Open Problems

We conjecture that our line formation protocol is optimal, i.e., no population protocol can construct a line containing all agents in parallel time $o(n \log n)$ whp. Going beyond the proposed strand self-replication protocol one could investigate whether other network structures can self-replicate and at what cost.

References

1 D. Angluin, J. Aspnes, Z. Diamadi, M.J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. In *Proc. PODC 2004*, pages 290–299, 2004.
2 D. Angluin, J. Aspnes, and D. Eisenstat. Fast computation by population protocols with a leader. *Distributed Comput.*, 21(3):183–199, 2008.
3 L. Gąsieniec, P.G. Spirakis, and G. Stachowiak. New clocks, optimal line formation and efficient replication population protocols (making population protocols alive). *CoRR*, abs/2111.10822, 2021. `arXiv:2111.10822`.
4 A. Czumaj and A. Lingas. On truly parallel time in population protocols. *CoRR*, abs/2108.11613, 2021. `arXiv:2108.11613`.
5 D. Doty, M. Eftekhari, L. Gąsieniec, E.E. Severson, G. Stachowiak, and P. Uznański. A time and space optimal stable population protocol solving exact majority. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021*, pages 1044–1055, 2021.
6 L. Gąsieniec and G. Stachowiak. Enhanced phase clocks, population protocols, and fast space optimal leader election. *J. ACM*, 68(1):2:1–2:21, 2021.
7 R.A. Freitas Jr and R.C. Merkle. *Kinematic Self-Replicating Machines*. Landes Bioscience, Georgetown, TX, 2004.
8 T.A. Lincoln and G.F. Joyce. Self-sustained replication of an rna enzyme. In *Science, Vol 323, Issue 5918*, pages 1229–1232. American Association for the Advancement of Science, 2009.
9 O. Michail and P. Spirakis. Simple and efficient local codes for distributed stable network construction. *Distributed Computing*, 29(3):207–237, 2016.
10 E. Moulin and N. Giuseppone. Dynamic combinatorial self-replicating systems. In *Constitutional Dynamic Chemistry*, pages 87–105. Springer, 2011.

Brief Announcement: Performance Anomalies in Concurrent Data Structure Microbenchmarks

Rosina F. Kharal
University of Waterloo, Canada

Trevor Brown
University of Waterloo, Canada

Abstract

Recent decades have witnessed a surge in the development of concurrent data structures with an increasing interest in data structures implementing concurrent sets (CSets). Microbenchmarking tools are frequently utilized to evaluate and compare performance differences across concurrent data structures. The underlying structure and design of the microbenchmarks themselves can play a hidden but influential role in performance results. However, the impact of microbenchmark design has not been well investigated. In this work, we illustrate instances where concurrent data structure performance results reported by a microbenchmark can vary 10-100x depending on the microbenchmark implementation details. We investigate factors leading to performance variance across three popular microbenchmarks and outline cases in which flawed microbenchmark design can lead to an inversion of performance results between two concurrent data structure implementations. We further derive a prescriptive approach for best practices in the design and utilization of concurrent data structure microbenchmarks.

2012 ACM Subject Classification Computing methodologies → Parallel computing methodologies

Keywords and phrases concurrent microbenchmarks, concurrent data structures, high performance simulations, PRNGs

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.45

Related Version *Full Version*: https://arxiv.org/abs/2208.08469

Funding This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) Collaborative Research and Development grant: CRDPJ 539431-19, the Canada Foundation for Innovation John R. Evans Leaders Fund with equal support from the Ontario Research Fund CFI Leaders Opportunity Fund: 38512, NSERC Discovery Program Grant: RGPIN-2019-04227, and the University of Waterloo.

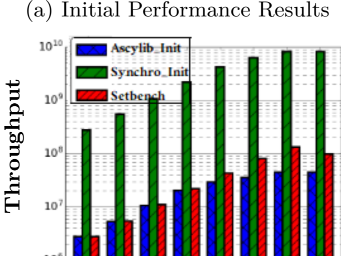
1 Introduction

An extensive variety of concurrent data structures have appeared over the past decade, with a particular focus on data structures implementing concurrent sets (CSets). A CSet is an abstract data type (ADT) which stores keys and provides three primary operations on keys: search, insert, and delete. Insert and delete operations modify the CSet and are called *update* operations. There are numerous concurrent data structures that can be used to implement CSets, including trees, skip-lists, and linked-lists. Microbenchmarks are commonly used to evaluate the performance of CSet data structures, essentially performing a stress test on the CSet across varying search/update workloads and thread counts. A typical microbenchmark runs an experimental loop bombarding the CSet with randomized operations performed by threads until the duration of the experiment expires. *Throughput*, number of operations performed by a CSet, is a key performance metric.

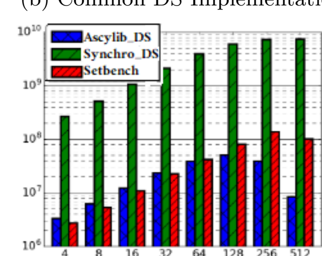
Multiple microbenchmarks exist to support CSet research. While CSet implementations have been well studied [1-3], the popular microbenchmarks used to evaluate them have not been scrutinized to the same degree. Microbenchmarking idiosyncrasies exist that can significantly distort performance results across data structures.

© Rosina F. Kharal and Trevor Brown;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 45; pp. 45:1–45:3
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

(a) Initial Performance Results



(b) Common DS Implementation



(c) Updates to Benchmarks

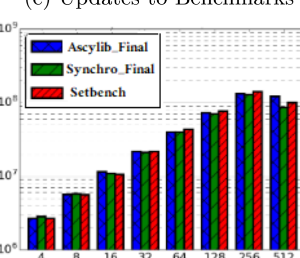


Figure 1 Throughput results across three microbenchmarks, Ascylib, Synchrobench and Setbench executing on a 256-core system testing the lock-free BST [5]. Results displayed on a logarithmic y-axis. Figure (a) results from unmodified microbenchmarks (as written by their authors). Figure (b) equalize the data structure (DS) implementations only. Ascylib_DS and Synchro_DS are updated with the imported lock-free BST implementation from Setbench. Figure (c) results for modified versions of Synchrobench and Ascylib correcting for pitfalls in microbenchmark design.

2 Microbenchmark Idiosyncrasies

When testing a CSet implementation on three different microbenchmarks with identical parameters, one would expect to observe similar performance results within a reasonable margin of error. However, we found 10-100x performance differences on the same CSet data structure tested across the Setbench [2], Ascylib [3], and Synchrobench [4] microbenchmarks. These microbenchmarks are often employed for evaluation of high performance CSets.

In Figure 1(a) we observe a range of varying performance results on the popular lock-free BST by Natarajan et al. [5] across the three microbenchmarks displayed using a logarithmic y-axis in order to capture wide performance gaps on a single scale. We performed a systematic review of the design intricacies within each microbenchmark. Our investigations led to the discovery that seemingly minor differences in the architecture and experimental design of a microbenchmark can cause a 10-100x performance boost erroneously indicating high performance of the data structure when the underlying cause is the microbenchmark itself. We perform successive updates to two of the microbenchmarks adjusting where errors or discrepancies were discovered until performance is approximately equalized (Figure 1(c)). In previous work by Arbel et al. [1], it was noted that *data structure implementation* differences can account for varying performance results in microbenchmark experimentation. We adjusted each microbenchmark to use a common lock-free BST implementation (Figure 1(b)) and still observed large performance gaps. Adjustments to the microbenchmark design were necessary to equalize results.

During our investigations, we found the following factors have the greatest impact on microbenchmark performance: (1) Repeated benchmark code is prone to error. In Synchrobench where the algorithm running performance experiments is duplicated for each data structure, errors in the algorithm led Synchrobench results to exceed other microbenchmarks by 100x. The microbenchmark testing algorithm should exist in one centralized location and provide easy adaptation to new data structures. (2) Pseudo random number generators (PRNGs) are typically used to generate random keys and operations. The way that a PRNG is integrated into the microbenchmark can play a significant role in experimental results. We investigate in detail in the full paper. (3) Microbenchmarks use a variety of techniques for splitting the update rate between insert and delete operations. For example, alternating between update operations, or flipping a biased coin to decide if

the next update will be an insert or delete. This has a non-trivial impact on performance. Recommended practice is to randomly distribute update operations between insert and delete operations using per thread PRNGs. (4) Synchrobench introduced a setting to enforce a specified rate of *effective* updates. An update operation is considered *effective* if it successfully modifies the CSet. Enforcing effective updates is problematic because, for example, in an almost full data structure, to perform an effective insert, one may need to repeatedly attempt to insert many random keys until one succeeds. The attempts leading up to the successful insert are essentially searches (which are faster than updates), and they are counted towards throughput, inflating performance. (5) Memory reclamation can play an influential role in performance results. The Ascylib microbenchmark memory reclamation algorithm is leaking memory at higher thread counts. This may render some microbenchmark experiments impracticable due to growth in memory usage. (6) Our recommended best practice for microbenchmark design includes strategies to detect and mitigate errors in the microbenchmark. We certainly recommend a checksum validation in microbenchmark experiments: the sum of keys inserted minus the sum of keys deleted into the CSet *during* an experiment should equal the final sum of keys contained in the CSet *following* the experiment. In our work, adding checksum validation assisted in discovering microbenchmark and data structure implementation errors. (7) We recommend a data structure prefilling step that includes randomized insert and delete operations. This generates a prefilled CSet data structure with a more realistic configuration, as opposed to a CSet that is produced by insert-only operations.

2.1 PRNG Usage in Concurrent Microbenchmarks

PRNGs are heavily relied upon in microbenchmarks to generate randomized keys and/or select randomized operations on a CSet. Researchers may be tempted to pre-generate a large array of random numbers prior to an experiment, thereby moving the cost of generating high quality randomness into the unmeasured setup phase of the experiment. We found this approach to be counter productive due to the impact on processor caching. In our full paper, we examine various PRNG methodologies and make practical recommendations for generating fast, high quality randomness, using hardware support where available.

References

1. Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. Getting to the root of concurrent binary search tree performance. In *2018 USENIX Annual Technical Conference*, 2018.
2. Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. Non-blocking interpolation search trees with doubly-logarithmic running time. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020.
3. Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.
4. Vincent Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 1–10, 2015.
5. Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, 2014.

Brief Announcement: Gathering Despite Defected View

Yonghwan Kim
Nagoya Institute of Technology, Aichi, Japan

Masahiro Shibata
Kyushu Institute of Technology, Fukuoka, Japan

Yuichi Sudo
Hosei University, Tokyo, Japan

Junya Nakamura
Toyohashi University of Technology, Aichi, Japan

Yoshiaki Katayama
Nagoya Institute of Technology, Aichi, Japan

Toshimitsu Masuzawa
Osaka University, Japan

Abstract

In this paper, we provide a new perspective on the observation by robots; a robot cannot necessarily observe all other robots regardless of distances to them. We introduce a new computational model with defected views called a (N,k)-defected model where k robots among $N-1$ other robots can be observed. We propose two gathering algorithms: one in the adversarial (N,$N-2$)-defected model for $N \geq 5$ (where N is the number of robots) and the other in the distance-based (4,2)-defected model. Moreover, we present two impossibility results for a (3,1)-defected model and a relaxed (N, $N-2$)-defected model respectively. This announcement is short; the full paper is available at [1].

2012 ACM Subject Classification Computing methodologies → Self-organization

Keywords and phrases mobile robot, gathering, defected view model

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.46

Related Version *Full Version*: https://arxiv.org/abs/2208.08159

Funding This work was supported in part by JSPS KAKENHI Grant Numbers 18K18031, 19H04085, 19K11823, 20H04140, 20KK0232, 21K17706, and Foundation of Public Interest of Tatematsu.

1 Introduction

An autonomous mobile robot system is a distributed system consisting of many mobile computational entities (called *robots*) with limited capabilities. Each robot observes the other robots (*Look*), computes the destination based on the observation result (*Compute*), and moves to the destination point (*Move*). Each robot autonomously and cyclically performs the above three operations to achieve the given common goal. Since an autonomous mobile robot system is firstly introduced in [2], many researchers are interested in clarifying the relationship between the capabilities of the robots and solvability of the problems.

Generally, in *Look* operation, each robot can observe all other robots to compute the destination point to move. In other words, each robot takes a snapshot consisting of all other robots' (relative) positions in its *Look* operation. However, from several practical reasons (e.g., memory restriction, memory corruption, or sensing failure), the positions of all robots may not be available necessarily available in *Compute* operation. This raises the main question we address: "*what occurs if a robot cannot observe some of other robots?*". More precisely, "*how many other robots should be observed to achieve the goals of the problems?*".

© Yonghwan Kim, Masahiro Shibata, Yuichi Sudo, Junya Nakamura, Yoshiaki Katayama, and Toshimitsu Masuzawa;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 46; pp. 46:1–46:3
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

To provide some answers for the above research questions, we propose a new computational model with restriction on the number of the robots that each robot can observe, named the *defected view model*, where each robot observes only k other robots for $1 \leq k < N-1$, where N is the number of robots. It is obvious that when k becomes the lower, the problem becomes the harder (possibly impossible) to solve. We consider two different defected view models regarding which k robots are observed: the adversarial (N,k)-defected model and the distance-based (N,k)-defected model (see Definition 1 for details).

As the first step of the study on the defected view model, we address the gathering problem and get the following results: two gathering algorithms in the adversarial (N, $N-2$)-defected model for $N \geq 5$ and the distance-based (4,2)-defected model, and some impossibility results to show the necessity of the assumptions the above algorithms use.

2 Model

Let $R = \{r_1, r_2, ..., r_N\}$ be the set of N autonomous mobile robots deployed in a plane. Robots are identical, uniform, oblivious, and have no geometrical agreement; they do not agree on any axis, the unit distance, nor chirality. A point in the plane is *occupied* if there exists a robot at the point. We allow two or more robots to occupy the same point at the same time. We call a robot a *single robot* if the point occupied by the robot has no other robot. Otherwise, we call it an *accompanied robot.* Each robot cyclically and synchronously performs the three operations, *Look*, *Compute*, and *Move*, we call the time duration in which all robots perform the three operations once *a round.* Moreover, we assume an unlimited visibility range and a *weak multiplicity detection.*

▶ **Definition 1** ((N,k)-defected model)**.** *Each robot r can get from* Look *operation the set of occupied points (in its coordinate system) where k robots not accompanied with r are located (i.e., the k robots contains no robot located at r's current point). When the number of robots not accompanied with r is less than k, all such robots are observed. The weak multiplicity detection concerning the k robots is assumed: a point occupied by only one of the k robots can be distinguished from that occupied by two or more of the k robots. Moreover, r can distinguish whether r is single or accompanied.*

We consider two options of the defected view model; *adversarial (N,k)-defected model* and *distance-based (N,k)-defected model.* In the *adversarial (N,k)-defected model*, k robots observed by each robot are determined adversarially. In the *distance-based (N,k)-defected model*, each robot r observes the k closest robots to the r's current point. Tie breaks among the robots the same distance apart is determined in an arbitrary way. In this paper, we consider the *Gathering Problem* to locate all robots at the same point under these models.

3 Proposed Algorithms and Impossiblity Results

Algorithm 1 presents an algorithm to achieve the gathering for robot r_i in the adversarial (N, $N-2$)-defected model where $N \geq 5$: OPSET() is a function that returns a set of points $\{p \mid p$ is occupied by r_i or by the robots that r_i observed$\}$, and isMulti(p) returns TRUE if point p is occupied by two or more robots that r_i observed (weak multiplicity), otherwise FALSE. The following theorem holds (we omit the proof).

▶ **Theorem 2.** *In the adversarial (N, $N-2$)-defected model ($N \geq 5$), Algorithm 1 solves the gathering problem in three rounds.*

Algorithm 1 Algorithm for robot r_i in the adversarial $(N, N-2)$-defected model where $N \geq 5$.

```
1: if ∀p ∈ OPSET() : isMulti(p) = TRUE then
2:     move to the center of the smallest enclosing circle of OPSET()
3: else if (r_i is single) ∧ (∃p ∈ OPSET() : isMulti(p) = TRUE) then
4:     move to an arbitrary point p ∈ OPSET() such that isMulti(p) = TRUE
5: else if ∀p ∈ OPSET() : isMulti(p) = FALSE then
6:     move to the center of the smallest enclosing circle of OPSET()
7: end if        ▷ No action if (r_i is accompanied) ∧ (∃p ∈ OPSET() : isMulti(p) = FALSE)
```

Algorithm 2 Gathering algorithm for robot r_i in the distance-based (4,2)-defected model.

```
1: if ∀p ∈ OPSET() : isMulti(p) = TRUE then
2:     move to the center of the smallest enclosing circle of OPSET()
3: else if (r_i is single) ∧ (∃p ∈ OPSET() : isMulti(p) = TRUE) then
4:     move to an arbitrary point p ∈ OPSET() such that isMulti(p) = TRUE
5: else if ∀p ∈ OPSET() : isMulti(p) = FALSE then
6:     if OPSET() forms an equilateral triangle then
7:         move to the center of the triangle (i.e., incenter)
8:     else if OPSET() forms an isosceles triangle then
9:         move to the midpoint of the base of the triangle
10:    else                          ▷ the other triangle or collinear three points
11:        move to the midpoint of the longest line
12:    end if
13: end if       ▷ No action if (r_i is accompanied) ∧ (∃p ∈ OPSET() : isMulti(p) = FALSE)
```

We do not know whether the gathering problem in the adversarial (4,2)-defected model is solvable or not yet. However, the gathering problem in the distance-based (4,2)-defected model can be solved by Algorithm 2 (Theorem 3). Moreover, there is no (deterministic) algorithm to solve the gathering problem in the defected view model for $N = 3$ (Theorem 4).

▶ **Theorem 3.** *In the distance-based (4, 2)-defected model, Algorithm 2 solves the gathering problem in four rounds.*

▶ **Theorem 4.** *There is no (deterministic) algorithm to solve the gathering problem in the distance-based (3,1)-defected model.*

The (N, k)-defected model assumes that k robots observed by robot r are chosen from the robots that are not accompanied with r and that r can detect whether it is single or accompanied. Natural relaxation of the model is to choose the k robots other than r (i.e., robots at r's current position can be chosen) and assume the weak multiplicity detection for the k robots and r itself. We call the model with the relaxation the *relaxed adversarial* (N,k)-defected model. The following impossibility result holds.

▶ **Theorem 5.** *There is no (deterministic) algorithm to solve the gathering problem in the relaxed adversarial $(N,N-2)$-defected model.*

References

1 Yonghwan Kim et al. Gathering despite defected view, 2022. doi:10.48550/ARXIV.2208.08159.

2 Ichiro Suzuki and Masafumi Yamashita. Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. *SIAM J. Comput.*, 28(4):1347–1363, 1999.

Brief Announcement: An Effective Geometric Communication Structure for Programmable Matter

Irina Kostitsyna ✉
TU Eindhoven, The Netherlands

Tom Peters ✉
TU Eindhoven, The Netherlands

Bettina Speckmann ✉
TU Eindhoven, The Netherlands

Abstract

The concept of programmable matter envisions a very large number of tiny and simple robot particles forming a smart material that can change its physical properties and shape based on the outcome of computation and movement performed by the individual particles in a concurrent manner. We use geometric insights to develop a new type of shortest path tree for programmable matter systems. Our *feather trees* utilize geometry to allow particles and information to traverse the programmable matter structure via shortest paths even in the presence of multiple overlapping trees.

2012 ACM Subject Classification Computing methodologies → Self-organization

Keywords and phrases Programmable matter, amoebot model, shape reconfiguration

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.47

Related Version *Full Version*: https://arxiv.org/abs/2202.11663

1 Introduction

The concept of programmable matter envisions a very large number of tiny and simple robot particles forming a smart material that can change its physical properties and shape based on the outcome of computation and movement performed by the individual particles in a concurrent manner. We focus on the *amoebot* model, which was introduced in [2] and refined in [1]. This model assumes a very small size of the particles and greatly restricts their computation, communication, and movement capabilities.

In the amoebot model particles occupy nodes of a triangular grid G embedded in the plane. A particle can occupy one (contracted particle) or two (expanded particle) adjacent nodes of the grid. The particles have limited computational power due to constant memory space, no common notion of orientation (disoriented), and no common notion of clockwise or counter-clockwise order (no consensus on chirality). They are identical (no IDs and they all execute the same algorithm), but can locally distinguish between their neighbors using six (for contracted particles) or ten (for expanded particles) *port identifiers*. Ports are labeled in order (either clockwise or counterclockwise) modulo six or ten, respectively. Particles communicate by sending messages to their neighbors using the ports and we assume a particle knows which of its neighbors ports is pointing to itself.

We call the set of particles and their internal states a *particle configuration* $\mathcal{P}$. Let $G_{\mathcal{P}}$ be the subgraph of G induced by the nodes occupied by particles in $\mathcal{P}$. We say that $\mathcal{P}$ is *connected* if there is a path in $G_{\mathcal{P}}$ between any two particles in $\mathcal{P}$. A *hole* in $\mathcal{P}$ is an interior face of $G_{\mathcal{P}}$ with more than three vertices. A particle configuration $\mathcal{P}$ is *simply connected* if it is connected and has no holes.

© Irina Kostitsyna, Tom Peters, and Bettina Speckmann;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 47; pp. 47:1–47:3
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 Shortest path trees

Among the previously proposed primitives for amoebot coordination is the *spanning forest primitive* [3] which organizes all particles into trees to facilitate movement while preserving connectivity. However, the spanning forest primitive does not impose any additional structure on the resulting spanning trees. We propose a type of shortest path trees, which we call *feather trees*, in which the path from any particle p to the root r of the feather tree is not longer than any unrestricted path from p to r in $G_{\mathcal{P}}$. Feather trees can be constructed as fast as spanning forests, namely in time linear in the diameter d of $G_{\mathcal{P}}$. They can be used for the same purposes, but have additional geometric properties that support efficient communication and movement of particles even in the presence of multiple overlapping trees.

2.1 Feather trees

We construct a feather tree from a root r in the following way. We distinguish between particles lying on *shafts* (emanating from the root r or other specific nodes) and *branches* (see Fig. 1 (left)). Each particle stores the direction of its parent and whether it is on a shaft or a branch. The root r chooses a maximum independent set of neighbors N_{ind}; the set contains at most three particles and there are at most two ways to choose it. The particles in N_{ind} form the bases of shafts emanating from r. All other neighbors of r form the bases of branches emanating from r. Specifically, let particle p be a neighbor of r across port i. The parent of p is set to $i + 3$ (recall that arithmetic on ports is modulo six), translated to the coordinate system of p. Particle p lies on a shaft if it is in N_{ind}, and on a branch otherwise.

A shaft particle with a parent in direction i propagates the shaft straight to the particle at $i + 3$, and branches into the two directions at ports $i + 2$ and $i + 4$. A branch particle with a parent in direction i propagates the branch straight to the particle at $i + 3$.

We say a *bend* in a path is formed by three consecutive vertices that form a 120° angle. By growing a feather tree according to the rules described so far, we process only particles that are reachable from r by a path with a single bend. We hence need to extend our construction around reflex vertices on the boundary of $\mathcal{P}$ that lie on branches. Specifically, if a branch particle p has a parent at direction i, the direction $i + 1$ (or $i - 1$) does not contain a particle, and the direction $i + 2$ (or $i - 2$) does contain a particle, then p initiates a growth of a new shaft in direction $i + 2$ (or $i - 2$) (see Fig. 1 (right)). We hence have the following lemma:

▶ **Lemma 1.** *Given a simply connected particle configuration $\mathcal{P}$ with diameter d, and a particle $r \in \mathcal{P}$, we can grow a feather tree from r in $O(d)$ rounds.*

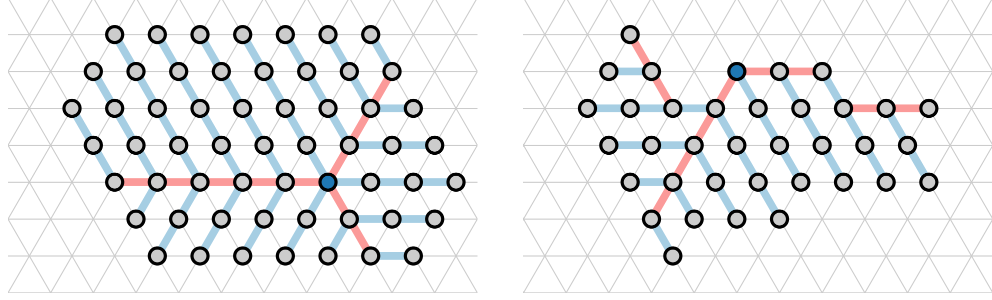


Figure 1 Two feather trees growing from the dark blue root. Shafts are red and branches are blue. Left: every particle is reachable by the initial feathers; Right: additional feathers are necessary.

Every particle is reached by a feather tree exactly once, from one particular direction. Hence, the feather tree is unique, independent of the activation sequence of the particles. In the following we describe how to navigate a set of overlapping feather trees. To do so, we first identify a useful property of shortest paths in feather trees.

We say that a vertex v of $G_\mathcal{P}$ is an *inner vertex*, if v and its six neighbors are part of $G_\mathcal{P}$. All other vertices are *boundary vertices*. We say that a bend is an *inner bend* if its middle vertex is an inner vertex; otherwise the bend is a *boundary bend*.

▶ **Definition 2** (Feather Path). *A path in $G_\mathcal{P}$ is a* feather path *if it does not contain two consecutive inner bends.*

The next lemma follows from the fact that inner bends can occur only on shafts, and any path must alternate visiting shafts and branches.

▶ **Lemma 3.** *Every path from the root to a leaf in a feather tree is a feather path.*

3 Communicating over shortest path trees

Consider a token t traversing a single feather tree F in a network of overlapping feather trees. Due to their limited memory, particles cannot store the identity of F. Despite that, due to Lemma 3, particles can propagate tokens down a feather tree by simply counting the number of inner bends. Thus, when a token is traversing a feather tree F down from the root, it always reaches a leaf of F via a shortest path through $\mathcal{P}$. In particular, it is always a valid choice for p to propagate t straight ahead (if feasible). A left or right 120° turn is a valid choice if it is a boundary bend, or if the last bend the token made was a boundary bend. We cannot control which leaf of F the token t reaches, but it does so without leaving F and hence along a shortest path. We can also broadcast token t to all leaves of F.

Consider now a node ℓ, which is a leaf of multiple feather trees. A token t starting out at node ℓ will always reach the root of one of its feather trees along a shortest path. However, we cannot control which root t reaches. Alternatively, we can broadcast t to all roots of the trees containing node ℓ. As before, the token t navigates by keeping track of the number of inner bends. In particular, if t already made one inner bend since its last boundary bend, then the only valid choice is to continue straight ahead. Otherwise, all three options (straight ahead or a 120° left or right turn) are valid.

References

1. Joshua J. Daymude, Andréa W. Richa, and Christian Scheideler. The Canonical Amoebot Model: Algorithms and Concurrency Control. In *35th International Symposium on Distributed Computing (DISC)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:19, 2021. doi:10.4230/LIPIcs.DISC.2021.20.
2. Zahra Derakhshandeh, Shlomi Dolev, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Brief announcement: Amoebot—A New Model for Programmable Matter. In *Proc. 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 220–222, 2014. doi:10.1145/2612669.2612712.
3. Zahra Derakhshandeh, Robert Gmyr, Thim Strothmann, Rida Bazzi, Andréa W. Richa, and Christian Scheideler. Leader Election and Shape Formation with Self-organizing Programmable Matter. In *Proc. International Workshop on DNA-Based Computing (DNA)*, LNCS 9211, pages 117–132, 2015. doi:10.1007/978-3-319-21999-8_8.

Brief Announcement: Distributed Quantum Interactive Proofs

François Le Gall ✉
Graduate School of Mathematics, Nagoya University, Japan

Masayuki Miyamoto ✉
Graduate School of Mathematics, Nagoya University, Japan

Harumichi Nishimura ✉
Graduate School of Informatics, Nagoya University, Japan

Abstract

The study of distributed interactive proofs was initiated by Kol, Oshman, and Saxena [PODC 2018] as a generalization of distributed decision mechanisms (proof-labeling schemes, etc.), and has received a lot of attention in recent years. In distributed interactive proofs, the nodes of an n-node network G can exchange short messages (called certificates) with a powerful prover. The goal is to decide if the input (including G itself) belongs to some language, with as few turns of interaction and as few bits exchanged between nodes and the prover as possible. There are several results showing that the size of certificates can be reduced drastically with a constant number of interactions compared to non-interactive distributed proofs.

In this brief announcement, we introduce the quantum counterpart of distributed interactive proofs: certificates can now be quantum bits, and the nodes of the network can perform quantum computation. The main result of this paper shows that by using quantum distributed interactive proofs, the number of interactions can be significantly reduced. More precisely, our main result shows that for any constant k, the class of languages that can be decided by a k-turn classical (i.e., non-quantum) distributed interactive protocol with $f(n)$-bit certificate size is contained in the class of languages that can be decided by a 5-turn distributed quantum interactive protocol with $O(f(n))$-bit certificate size. We also show that if we allow to use shared randomness, the number of turns can be reduced to 3-turn. Since no similar turn-reduction *classical* technique is currently known, our result gives evidence of the power of quantum computation in the setting of distributed interactive proofs as well.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Quantum computation theory

Keywords and phrases distributed interactive proofs, distributed verification, quantum computation

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.48

Related Version *Full Version*: https://arxiv.org/abs/2210.01390

Funding FLG was supported by the JSPS KAKENHI grants JP16H01705, JP19H04066, JP20H00579, JP20H04139, JP20H05966, JP21H04879 and by the MEXT Q-LEAP grants JPMXS0118067394 and JPMXS0120319794. MM was supported by JST, the establishment of University fellowships towards the creation of science technology innovation, Grant Number JPMJFS2120. HN was supported by the JSPS KAKENHI grants JP19H04066, JP20H05966, JP21H04879, JP22H00522 and by the MEXT Q-LEAP grants JPMXS0120319794.

1 Introduction

In distributed computing, the topology of the communication network is fundamental information and efficient verification of graph properties of the network is useful from both theoretical and applied aspects. The study of this notion of verification in the distributed setting has lead to the notion of "distributed NP" in analogy with the complexity class NP in centralized computation: A powerful prover provides certificates to each node of the network

© François Le Gall, Masayuki Miyamoto, and Harumichi Nishimura;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 48; pp. 48:1–48:3
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

in order to convince that the network has a desired property; If the property is satisfied, all nodes must output “accept”, otherwise at least one node must output “reject”. This concept of “distributed NP” has been formulated in several ways, including *proof-labeling schemes* (PLS) [5], *non-deterministic local decision* (NLD) [2], and *locally checkable proofs* (LCP) [3].

As a motivating example, consider the problem of verifying whether the network is bipartite or not. While this problem cannot be solved in $O(1)$ round without prover, it can easily be solved with a prover telling to each node to each part it belongs to, which requires only a 1-bit certificate per node, and then each node broadcasting this information to its adjacent nodes (here the crucial point is that if the network is non-bipartite, then at least one node will be able to detect it). On the other hand, it is known that there exist properties that require large certificate size to decide: Göös and Suomela [3] have shown that recognizing symmetric graphs (Sym) and non 3-colorable graphs ($\overline{\text{3Col}}$) require $\Omega(n^2)$-bit certificates per node in the framework of LCP (which is tight since all graph properties are locally decidable by giving the $O(n^2)$-bit adjacency matrix of the graph).

To reduce the length of the certificate for such problems, the notion of distributed interactive proofs (also called distributed Arthur-Merlin proofs) was recently introduced by Kol, Oshman and Saxena [4] as a generalization of distributed NP. In this model there are two players, the prover (often called Merlin), who has unlimited computational power and sees the entire network but is untrusted (i.e., can be malicious), and the verifier (often called Arthur) representing all the nodes of the network, who can perform only local computation and brief communication with adjacent nodes. Generalizing the concept of distributed NP, the nodes are now allowed to engage in multiple turns of interaction with the prover. As for distributed NP, there are two requirements of the protocol: if the input is legal (yes-instance) then all nodes must accept with high probability (*completeness*), and if the input is illegal then at least one node must reject with high probability (*soundness*).

In the setting of [4], each node has access to a private source of randomness, and sends generated random bits to the prover in Arthur’s turn. For instance, a 2-turn protocol contains two interactions: Arthur first queries Merlin by sending a random string from each node, and then Merlin provides a certificate to each node. After that, nodes exchange messages with adjacent nodes to decide their outputs. The main complexity measures when studying distributed interactive protocols are the size of certificates provided to each node, the size of the random strings generated at each node and the size of the messages exchanged between nodes. Let us denote $\mathsf{dAM}[k](f(n))$ the class of languages that have k-turn distributed Arthur-Merlin protocols where Merlin provides $O(f(n))$-bit certificates, Arthur generates $O(f(n))$-bit random strings at each node and $O(f(n))$-bit messages are exchanged between nodes. Kol et al. [4] showed the power of interaction by giving a $\mathsf{dAM}[3](\log n)$ protocol for graph symmetry (Sym) and a $\mathsf{dAM}[4](n \log n)$ protocol for graph non-isomorphism (GNI), which are known to require $\Omega(n^2)$-bit certificate in LCP [3].

2 Our Results

In this paper we introduce the quantum counterpart of distributed interactive proofs, which we call distributed quantum interactive proofs (or sometimes distributed quantum interactive protocols) and write dQIP, and show their power. Roughly speaking, distributed quantum interactive proofs are defined similarly to the classical distributed interactive proofs (i.e., distributed Arthur-Merlin proofs) defined above, but the messages exchanged between the prover and the nodes of the network can now contain quantum bits (qubits), the nodes can now do any (local) quantum computation (i.e., each node can apply any unitary transform

to the registers it holds), and each node can now send messages consisting of qubits to its adjacent nodes. In analogy to the classical case, the main complexity measures when studying distributed quantum interactive protocols are the size of registers exchanged between the prover and the nodes, and the size of messages exchanged between the nodes. We give the formal definition of dQIP in the full version of our paper. The class $\mathsf{dQIP}[k](f(n))$ is defined as the set of all languages that can be decided by a k-turn dQIP protocol where both the size of the messages exchanged between the prover and the nodes, and the size of the messages exchanged between the nodes are $O(f(n))$ qubits.

Our first result is the following theorem.

▶ **Theorem 1.** *For any constant* $k \geq 1$, $\mathsf{dAM}[k](f(n)) \subseteq \mathsf{dQIP}[5](f(n))$.

Theorem 1 shows that by using distributed quantum interactive proofs, the number of interactions in distributed interactive proofs can be significantly reduced. To prove this result, we develop a generic *quantum* technique for turn reduction in distributed interactive proofs. Since no similar turn-reduction *classical* technique is currently known, our result gives evidence of the power of quantum computation in the setting of distributed interactive proofs as well.

We also show that if we allow to use randomness shared to all nodes (we denote this model by dQIP^{sh}), the number of turns can be further reduced to three turns.

▶ **Theorem 2.** *For any constant* $k \geq 1$, $\mathsf{dAM}[k](f(n)) \subseteq \mathsf{dQIP}^{sh}[3](f(n))$.

On the other hand, in the classical case, it is known that allowing shared randomness does not change the class [1]: $\mathsf{dAM}^{sh}[k](f(n)) \subseteq \mathsf{dAM}[k](f(n))$ for all $k \geq 3$ (in fact, the authors of [1] showed $\mathsf{dAM}^{sh}[k](f(n)) \subseteq \mathsf{dAM}[k](f(n) + \log n)$ for all $k \geq 1$ where the additional $\log n$ comes from constructing a spanning tree, but for $k \geq 3$, a spanning tree can be constructed with $O(1)$-sized messages between the prover and the nodes in three turns [6]).

As mentioned above, for (classical) dAM protocols increasing the number of turns is helpful to reduce the complexity (in particular, the certificate size) for many problems. Our result thus shows if we allow quantum resource, such protocols can be simulated in five turns, and in three turns if we allow shared randomness.

References

1. Pierluigi Crescenzi, Pierre Fraigniaud, and Ami Paz. Trade-Offs in Distributed Interactive Proofs. In *Proceedings of the 33rd International Symposium on Distributed Computing (DISC 2019)*, pages 13:1–13:17, 2019.
2. Pierre Fraigniaud, Amos Korman, and David Peleg. Local distributed decision. In *Proceedings of the IEEE 52nd Annual Symposium on Foundations of Computer Science (FOCS 2011)*, pages 708–717, 2011.
3. Mika Göös and Jukka Suomela. Locally checkable proofs in distributed computing. *Theory of Computing*, 12(1):1–33, 2016.
4. Gillat Kol, Rotem Oshman, and Raghuvansh R Saxena. Interactive distributed proofs. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing (PODC 2018)*, pages 255–264, 2018.
5. Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Computing*, 22(4):215–233, 2010.
6. Moni Naor, Merav Parter, and Eylon Yogev. The power of distributed verifiers in interactive proofs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2020)*, pages 1096–115, 2020.

Brief Announcement: Null Messages, Information and Coordination

Raïssa Nataf ✉
Technion, Haifa, Israel

Guy Goren ✉
Technion, Haifa, Israel

Yoram Moses ✉
Technion, Haifa, Israel

Abstract

This paper investigates how null messages can transfer information in fault-prone synchronous systems. The notion of an *f-resilient message block* is defined and is shown to capture the fundamental communication pattern for knowledge transfer. In general, this pattern combines both null messages and explicit messages. It thus provides a fault-tolerant extension of the classic notion of a message-chain. Based on the above, we provide tight necessary and sufficient characterizations of the generalized communication patterns that can serve to solve the distributed tasks of (nice-run) Signalling and Ordered Response.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Computing methodologies → Reasoning about belief and knowledge

Keywords and phrases null messages, fault tolerance, coordination, information flow

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.49

Related Version *Full Version*: https://arxiv.org/abs/2208.10866

Funding *Guy Goren*: Guy Goren was partly supported by a grant from the Technion Hiroshi Fujiwara cyber security research center and the Israel cyber bureau, as well as by a Jacobs fellowship. *Yoram Moses*: Yoram Moses is the Israel Pollak academic chair at the Technion. Both his work and that of Raïssa Nataf were supported in part by the Israel Science Foundation under grant 2061/19.

1 Introduction

In synchronous models with a global clock, it may be possible to transmit information by *not* sending a message, which Lamport termed *sending a null message* in [7]. While null messages have been successfully employed to optimize communication in useful protocols (see, e.g., [1, 6] for early examples), the question of how null messages convey information, and what information they convey, has only been partly addressed. In addition, when failures can occur, the use of null messages can become rather challenging. If i does not receive a message from j in such a setting, i might not be able to distinguish whether this is because j purposely refrained from sending, or because j failed. Nevertheless, recent work [4] has shown that when the number of failures is bounded (by f, say), it is still possible to use null messages to transmit information. Very roughly speaking, their "Silent Choir" theorem implies that in a failure-free execution, the only way that a process j can learn i's value without receiving an explicit message chain from i is for there to be a set of $f+1$ processes that received such a chain from i do not send a message to j. However, this is far from being sufficient. Our purpose in this paper is to initiate a systematic analysis of the role of null messages, and obtain sharper characterizations of their use when processes can fail.

© Raïssa Nataf, Guy Goren, and Yoram Moses;
licensed under Creative Commons License CC-BY 4.0

We consider the standard synchronous message-passing model with crash failures. We assume a finite set $\mathbb{P}$ of processes that are connected via a communication network, and all start at time 0. Moreover, messages are reliably delivered in one time step. We call one of the processes the "source," and denote it by s. For simplicity, we restrict our attention to the case in which s has a binary initial value $v_s \in \{0, 1\}$, while every process $i \neq s$ has a unique initial state (with a fixed initial value, say 0). We assume a bound of f on the number of processes that can crash in any given run. Finally, we focus on deterministic protocols, so a ***protocol*** Q describes what messages a process sends and what decisions it takes, as a function of its local state. In particular, a protocol Q has a single run in which $v_s = 1$ and no failures occur. We call this run Q's *nice run*, and denote it by $\hat{r}(Q)$, or simply by $\hat{r}$ when Q is clear from context. A process is said to be ***active*** at time m if it has not crashed by time $m - 1$ and it correctly follows its protocol at time m. For more about our formal model, see [8]. Our analysis makes use of a formal theory of knowledge to capture how null messages affect what processes do or do not learn. See [3] for more details and a general introduction to the topic.

When the model is synchronous, it is common to consider the event that i did not send its neighbor j a message at time t in a given run as if i sent j a null message there. Of course, j will be able to observe at time $t + 1$ that no message was received. Notice, however, that if i *never* sends j a message at time t then this will not provide j any information whatsoever. We say that i sends j a *genuine* null message at time t in a run r if, in addition, there is some run $r' \neq r$ in which i *does* send a message to j at time t. From here on, all null messages will assumed to be genuine. To capture the information conveyed by a null message, we use the following:

▶ **Definition 1** (Null message sent in case φ). *We say that in protocol Q process i sends a* null message *to j at time t* in case *φ if for every run r of Q in which i is active at time t, it* does not *send a message to j at time t in r if and only if φ holds at time t in r.*

Clearly, in a failure-free system (i.e., if $f = 0$) if i sends j a null message at time t then, at time $t + 1$ process j comes to know that φ was true. In the presence of failures, however, j is not guaranteed to know this, and a more subtle analysis is required. We begin by considering the problem of transmitting information about the initial value v_s of the source process to another process $j \neq s$. More precisely, we define a problem called *nice-run signalling* (NS) in the following manner. Following [2], we use the notation $\langle i, t\rangle$, which we call a *process-time node* to stand for process i at time t. A protocol Q is said to *solve* nice-run signalling (NS) between $\langle s, 0\rangle$ and $\langle j, m\rangle$ if $K_j(v_s = 1)$ holds at time m in Q's nice run $\hat{r}(Q)$. Instances of NS often appear when optimizing the communication costs of protocols that solve other distributed tasks (e.g., optimizing the good-case costs of Consensus [5]).

2 f-resilient message block

We now turn to study the communication patterns that protocols solving NS and related problems can use in their nice runs. We focus on "communication graphs," denoted by $CG_Q(r) = (\mathbb{V}, E_m, E_n, E_\ell)$, that account for the messages and the null messages that are sent in a run r of a given protocol Q. The set $\mathbb{V}$ of nodes of the graph consists of all process-time nodes $\theta = \langle i, t\rangle$, with $t \geq 0$. The set E_m consists of directed edges (θ, θ') such that a message is sent in r at θ and delivered to θ', while E_n consists of directed edges (θ, θ') such that a (genuine) null message is sent in r from θ to θ'. Finally, E_ℓ consists of all edges of the form $(\langle i, t\rangle, \langle i, t + 1\rangle)$, $i \in \mathbb{P}$ and $t \geq 0$, between consecutive nodes along the timeline of a process.

In general, a path in the communication graph $CG_Q(r)$ records a chain consisting of both actual messages and null messages. We therefore refer to it as a *weak* message chain. We are now ready to define a primitive that plays an essential role in solutions to NS.

▶ **Definition 2** (f-resilient message block). *Let $\theta, \theta' \in \mathbb{P} \times \mathbb{N}$ be two nodes. An f-*resilient message block from θ to θ' in $CG_Q(r)$ *is a set Γ of paths between θ and θ' such that for all sets $B \subset \mathbb{P}$ with $|B| \leq f$ there is a path in $CG_Q(r)$ that does not contain null messages sent by processes in B.*

Notice that a path that does not contain *null* messages sent by a process j can still contain messages sent by j. As a result, if the adversary crashes j, this path may still convey information. In a precise sense, f-resilient message blocks are both necessary and sufficient for solving nice-run signalling, and we can obtain a *tight* characterization of the communication patterns needed for solving NS:

▶ **Theorem 3.**

- (Necessity) *If a protocol Q solves* NS *from $\theta_s = \langle s, 0\rangle$ to $\theta_j = \langle j, m\rangle$, then there must be an f-resilient message block from θ_s to θ_j in $CG_Q(\hat{r})$. (Recall that $\hat{r}$ is Q's nice run.)*
- (Sufficiency) *If a communication graph CG contains an f-resilient message block between $\theta_s = \langle s, 0\rangle$ and $\theta_j = \langle j, m\rangle$, then there exists a protocol Q with $CG_Q(\hat{r}) = CG$. that solves* NS *between θ_s and θ_j.*

Beyond direct information transfer as captured by the NS problem, we proceed in [8] to consider a coordination problem called Ordered Response (OR) in which processes must perform actions in a linear temporal order as a reaction to a spontaneous event (See [2]). Namely, each process $i_h \in \{i_1, i_2, \ldots, i_k\}$ has a specific action a_h to perform, and these actions should be performed only if initially $v_s = 1$. Moreover, they need to be performed in temporal order. I.e., denoting by t_h the time at which i_h performs a_h, it is required that $t_1 \leq t_2 \leq \cdots \leq t_k$. Finally, we consider the variant in which all actions are performed in the nice run. One way to solve this while ensuring no OR violation in any run is, roughly speaking, to create f-resilient message blocks in $CG(\hat{r})$, i.e., solving NS between each consecutive pair of processes in the order, to inform i_{h+1} that i_h has acted. This would be governed by Theorem 3. However, information may also be transferred indirectly: for instance, if process i_3 knows that i_2 knows that $v_s = 1$ and that f processes – not including i_2 have failed – then it can infer that a_2 has been performed, and so i_3 can "safely" perform a_3. For characterizations of the communication patterns used in solutions to OR, see [8].

References

1. Eugene S. Amdur, Samuel M. Weber, and Vassos Hadzilacos. On the message complexity of binary byzantine agreement under crash failures. *Distributed Computing*, 5(4):175–186, 1992.
2. Ido Ben-Zvi and Yoram Moses. Beyond Lamport's happened-before: On time bounds and the ordering of events in distributed systems. *Journal of the ACM (JACM)*, 61(2):1–26, 2014.
3. Ronald Fagin, Joseph Y Halpern, Yoram Moses, and Moshe Y Vardi. *Reasoning About Knowledge*. MIT Press, 1995. doi:10.7551/mitpress/5803.001.0001.
4. Guy Goren and Yoram Moses. Silence. *J. ACM*, 67:3:1–3:26, 2020. doi:10.1145/3377883.
5. Guy Goren and Yoram Moses. Optimistically tuning synchronous Byzantine consensus: another win for null messages. *Distributed Computing*, 34(5):395–410, 2021.
6. Vassos Hadzilacos and Joseph Y. Halpern. Message-optimal protocols for byzantine agreement. *Mathematical Systems Theory*, 26(1):41–102, 1993.
7. Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst.*, 6:254–280, 1984. doi:10.1145/2993.2994.
8. Raïssa Nataf, Guy Goren, and Yoram Moses. Null messages, information and coordination: Preliminary report. *CoRR*, abs/2208.10866, 2022. arXiv:2208.10866.

Brief Announcement: Asymmetric Mutual Exclusion for RDMA

Jacob Nelson-Slivon ✉
Lehigh University, Betlehem, PA, USA

Lewis Tseng ✉
Boston College, MA, USA

Roberto Palmieri ✉
Lehigh University, Betlehem, PA, USA

Abstract

In this brief announcement, we define operation asymmetry, which captures how processes may interact with an object differently, and discuss its implications in the context of a popular network communication technology, remote direct memory access (RDMA). Then, we present a novel approach to mutual exclusion for RDMA-based distributed synchronization under operation asymmetry. Our approach avoids RDMA loopback for local processes and guarantees starvation-freedom and fairness.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Mutual exclusion, Synchronization, Remote direct memory access (RDMA)

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.50

Related Version *Technical Report*: `https://arxiv.org/abs/2208.09540`

Funding This material is based upon work supported by the National Science Foundation under Grant No. CNS-2045976.

1 Introduction

In contrast to traditional message-passing, remote direct memory access (RDMA) is a popular network communication technology that directly implements the shared-memory abstraction in the distributed setting by allowing a process to access memory on a remote machine *without* interacting with another process. These operations are known as *one-sided*, since they only involve one process. In addition to reads and writes, RDMA also provides atomic read-modify-write (RMW) operations on remote memory, like compare-and-swap (CAS) and fetch-and-add (FAA). Hence, the API closely resembles that of modern shared-memory.

Also similar to modern architectures, the memory semantics of RDMA is *not* sequentially consistent. Since remote operations complete asynchronously, local and remote access to a given memory location may be reordered. Furthermore, while remote reads and writes are atomic with their local counterparts due to cache coherent I/O (e.g., Intel's DDIO), atomicity between local and remote RMW operations is not guaranteed. Without *global atomicity* support (i.e., atomicity among all local and remote operations), all processes must rely on the RDMA-capable network interface controller (RNIC) for consistency. More precisely, local processes should use the *loopback* mechanism, which allows a process to access memory on its own machine by passing through the RNIC.

In practice, both one-sided RDMA RMW and message-passing (e.g., RPCs) have their drawbacks. For the one-sided approach, RDMA loopback is still an order of magnitude slower than local accesses and introduces internal congestion [2]. While RPCs are prevalent in RDMA-based systems, in part due to the many challenges associated with synchronizing local

© Jacob Nelson-Slivon, Lewis Tseng, and Roberto Palmieri;
licensed under Creative Commons License CC-BY 4.0

and remote processes, message-passing can nullify the performance benefits that one-sided RDMA provides. Thus, a primary motivation for our work is how to balance the needs of both local and remote processes in the context of one-sided RDMA.

To that end, this brief announcement introduces the concept of *operation asymmetry*, a property that captures how processes interact with memory differently, and describes a new mutual exclusion algorithm designed to capture the nuanced requirements of synchronizing local and remote processes in RDMA-enabled systems. To the best of our knowledge, we are the first to solve mutual exclusion specifically for RDMA in a manner that does *not* require RDMA loopback or message-passing. Our solution is *starvation-free* (i.e., a calling process eventually executes its critical section) and *fair* (i.e., first-come-first-served).

2 Mutual Exclusion Under Operation Asymmetry

In our system model, processes communicate by accessing *local* or *remote* shared memory, consisting of *registers*. For each class of access (local/remote), the registers in our system support three operations: read (`Read`/`rRead`), write (`Write`/`rWrite`) and compare-and-swap (`CAS`/`rCAS`). Local operations access memory natively while remote operations pass through the RNIC. An operation on a register is *enabled* for a process if the process is able to access the register using the given operation. Intuitively, local accesses are only enabled for local processes (i.e., on the same machine as the register).

We define an object as *operation asymmetric* if, given two processes, the intersection of their respective enabled operations on the object is *not equal* to their union. Under one-sided RDMA, registers are operation asymmetric since remote processes cannot perform local accesses. To demonstrate the consequences of operation asymmetry, recall that the atomicity of local and remote operations is not guaranteed. Due to this behavior, an RDMA RMW operation (e.g., `rCAS`) appears to a local process as if it were a `Read` then `Write`. Hence, local processes must utilize RDMA loopback to ensure atomicity of RMW operations with remote processes.

When designing a mutual exclusion primitive for RDMA-based systems (without global atomicity), remote RMW operations provide the necessary atomicity but RDMA loopback introduces overhead for local processes and network congestion. Therefore in our model, to avoid local processes using RDMA loopback, we restrict the set of enabled operations for local processes to *only include local operations.* However, due to operation asymmetry, any solution satisfying these constraints can only be built from the greatest common denominator: atomic read-write registers. Thus, approaches like Peterson's lock [5] are appropriate.

2.1 Algorithm Description

To implement multi-process synchronization using operation asymmetric registers, we modify the original (two-process) Peterson's lock algorithm to embed an orthogonal mutual exclusion primitive whereby local and remote processes compete amongst themselves for the right to participate in the Peterson's lock protocol. To limit the number of remote operations required for remote processes, we embed the widely used MCS queue lock [3], allowing processes to spin locally while waiting to acquire the lock. Our combination of locks is an extension of lock cohorting [1] to an RDMA-enabled distributed system, and we adopt the naming conventions by calling Peterson's lock the *global* lock and the MCS queue locks *cohort* locks. In our approach, processes with the same set of enabled operations (local or remote) compete amongst themselves using their cohort lock to determine a leader that then participates in the global protocol, relying only on process-wide atomic operations (i.e., read and write).

The original Peterson's lock algorithm has two global variables: `flag[2]`, which is a two element array of boolean values indicating interest in the critical section, and `victim`, which is an integer deciding which process yields execution. We modify the algorithm by replacing `flag` with our MCS queue cohort locks.

A process first announces interest in executing its critical section by locking the corresponding (local or remote) cohort lock, effectively raising its `flag`. If the calling process acquires the cohort lock from another member of its cohort, it may enter the critical section without additional steps. Otherwise, the process must engage in the (global) Peterson's lock protocol, by setting `victim` to its own process identifier then waiting while the other cohort lock is held and `victim` is not changed. Since Peterson's lock is constructed from atomic read-write registers, and local and remote reads and writes are atomic, local operations need only use local accesses, remote operations use one-sided RDMA, and no RDMA loopback is necessary. To unlock, a process simply releases its cohort lock, effectively lowering the `flag` variable of the original algorithm.

Each cohort lock is specifically designed for the class of processes in the cohort. That is, there is one for local processes and another for remote processes. Note that MCS queue locks are particularly well-suited for RDMA since they perform *local-spinning*, meaning that a process need not repeatedly access remote memory while waiting for the lock. To implement fairness, we alter our MCS queue algorithms to support a budget, similar to the technique used by Dice et al. [1]. Once the budget is exhausted, the detecting process is required to reacquire the global lock. If there is a waiting process of the opposite cohort, it will be allowed to proceed. Otherwise, the calling process reacquires the global lock then resets the budget. Since the global lock is released after a bounded number of cohort lock acquisitions, and the global lock is itself fair (i.e., a waiting process cannot be overtaken), our approach is fair [1]. Our technical report [4] includes more details, the pseudo-code, and a model-checked TLA+ specification of our mutual exclusion primitive.

3 Conclusion

Motivated by our definition of *operation asymmetry*, we propose a starvation-free and fair mutual exclusion mechanism for RDMA, enabling local and remote processes to synchronize while optimizing for their individual behavioral constraints. To the best of our knowledge, we present the first mutual exclusion solution that synchronizes local and remote processes while avoiding both RDMA loopback and message-passing.

References

1 David Dice, Virendra J. Marathe, and Nir Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. In *PPoPP '12*, pages 247–256, 2012. doi:10.1145/2145816.2145848.

2 Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding Performance Anomalies in RDMA Subsystems. In *NSDI '22*, pages 287–305, Renton, WA, 2022.

3 John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991. doi:10.1145/103727.103729.

4 Jacob Nelson-Slivon, Lewis Tseng, and Roberto Palmieri. Technical Report: Asymmetric Mutual Exclusion for RDMA, 2022. arXiv:2208.09540.

5 Gary L. Peterson. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 12:115–116, 1981.

Brief Announcement: Foraging in Particle Systems via Self-Induced Phase Changes

Shunhao Oh ✉
School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA

Dana Randall ✉
School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA

Andréa W. Richa ✉
School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ, USA

Abstract

The *foraging problem* asks how a collective of particles with limited computational, communication and movement capabilities can autonomously compress around a food source and disperse when the food is depleted or shifted, which may occur at arbitrary times. We would like the particles to iteratively self-organize, using only local interactions, to correctly *gather* whenever a food particle remains in a position long enough and *search* if no food particle has existed recently. Unlike previous approaches, these search and gather phases should be self-induced so as to be indefinitely repeatable as the food evolves, with microscopic changes to the food triggering macroscopic, system-wide phase transitions. We present a stochastic foraging algorithm based on a phase change in the fixed magnetization Ising model from statistical physics: Our algorithm is the first to leverage *self-induced phase changes* as an algorithmic tool. A key component of our algorithm is a careful token passing mechanism ensuring a dispersion broadcast wave will always outpace a compression wave.

2012 ACM Subject Classification Theory of computation → Self-organization; Theory of computation → Random walks and Markov chains

Keywords and phrases Foraging, self-organized particle systems, compression, phase changes

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.51

Related Version *Full Version*: http://arxiv.org/abs/2208.10720

Funding *Shunhao Oh*: NSF award CCF-1733812 and ARO MURI award W911NF-19-1-0233
Dana Randall: NSF awards CCF-1733812 and CCF-2106687 and ARO MURI award W911NF-19-1-0233
Andréa W. Richa: NSF awards CCF-1733680 and CCF-2106917 and ARO MURI award W911NF-19-1-0233

1 The Foraging Problem

Collective behavior of interacting agents is a fundamental, nearly ubiquitous phenomenon across fields, reliably producing rich and complex coordination. Examples at the micro- and nano-scales include coordinating cells (including our own immune system or self-repairing tissue and bacterial colonies), micro-scale swarm robotics, and interacting particle systems in physics; at the macro scale it can represent flocks of birds, coordination of drones, and societal dynamics such as segregation. Common properties of many of these disparate systems is that they 1) respond to simple environmental conditions and 2) undergo *phase changes* as parameters of the systems are slowly modified, allowing collectives to gracefully toggle between two often dramatically different macroscopic states.

In the *foraging problem*, we consider a collective of "ants" (i.e., *particles*) with limited computational, communication and movement capabilities that reside on the triangular lattice, along with a *food* particle (i.e., any resource in the environment, e.g., an energy source) that may be placed at any point, removed, or shifted at arbitrary times, possibly adversarially. We would like the particles to consistently self-organize, using only local

© Shunhao Oh, Dana Randall, and Andréa W. Richa;
licensed under Creative Commons License CC-BY 4.0

36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 51; pp. 51:1–51:3
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

interactions, such that if a food particle remains in a position long enough, the particles should transition to a *gather phase* in which many collectively form a single large component with small perimeter around the food. Alternatively, if no food particle has existed recently, the particles should undergo a self-induced phase change and switch to a *search phase* in which they distribute themselves randomly throughout the lattice region to search for food. Unlike previous approaches, this process should be indefinitely repeatable, withstanding overlapping waves of phase changes that may interfere with each other. Like a physical phase change, microscopic changes such as the deletion or addition of a single food particle should trigger these macroscopic, system-wide transitions. This foraging problem has several fundamental application domains, including search-and-rescue operations in swarms of nano- or micro-robots; health applications (e.g., a collective of nano-sensors that could search for, identify, and gather around a foreign body to isolate or consume it, then resume searching, etc.); and finding and consuming/deactivating hazards in a nuclear reactor or a minefield.

2 Model and Preliminaries

In this work, we consider an abstraction of a self-organizing particle system (SOPS), where particles sit on vertices of a finite region of the triangular lattice. We assume particles have constant-size memory, but lack global orientation or any other global information beyond a common chirality. Particles communicate by sending tokens to their nearest neighbors in the lattice, where a *token* is a constant-size piece of information. Individual particles are activated according to their own Poisson clocks, possibly with different rates, and perform instantaneous actions upon activation. Particles are aware of their own and their neighbors' current states and when a particle is activated, it may do a bounded amount of computation, send at most one token (not necessarily identical) to each of its neighbors, and choose one of its six neighbors in the lattice to see if it is unoccupied and move there.[1]

Cannon et al. [1] introduced a related non-adaptive compression and expansion algorithm based on an input parameter λ that defines *system-wide behavior*. When λ is sufficiently small, the system is in an expansion phase, desirable to search for food, while when λ is large, the system will be in a compression phase, desirable when food has been discovered.[2] More specifically, using insight from the Ising model in statistical physics, the authors proved that adding a ferromagnetic attraction λ between particles suffices to stochastically lead the particles in a SOPS to an α-*compressed* configuration with high probability, where the constant $\alpha > 1$ determines an upper bound on the ratio of the configuration perimeter by the minimum possible system perimeter. The Markov chain is defined so that each configuration σ appears with probability $\pi(\sigma) = \lambda^{|E(\sigma)|}/Z$ at stationarity, where $|E(\sigma)|$ is the number of edges in σ and Z is the normalizing constant. It is rigorously shown that the SOPS will reach an α-compressed configuration at stationarity, for some constant $\alpha > 1$, if the attraction force λ is strong enough. Moreover, it is also shown that when the attraction forces are small, the configurations will nearly maximize their perimeter and *disperse* if particles are allowed to disconnect [2], as we do in our algorithm.

Our challenge here is to *self-induce* these system-wide behaviors upon the discovery or depletion of a single food particle. When food is not present, particles communicate to transition to the search phase by collectively lowering λ, and when food is discovered they transition to the gather phase, collectively raising λ to compress around the food.

[1] Our model can be seen as an abstraction of the (canonical) *Amoebot model* under a *sequential scheduler*.
[2] A similar algorithm for the more general setting where particles are allowed to disconnect also provably exhibits a bifurcation, but the notion of compression becomes more complicated [2].

3 The Adaptive Foraging Algorithm

We present the first rigorous local distributed algorithm for solving the foraging problem, the *Adaptive α-Compression algorithm.* There are two main (micro-level) states each particle can be in at any point in time, *dispersion* or *compression*, corresponding to the macro-level *search* and *gather* phases respectively. To switch to the search phase, particles are induced to collectively transition to the dispersion state. Likewise, to switch to gather, particles are induced to transition towards compression. Particles in the dispersion state move around in a process akin to a simple exclusion process, where they perform a random walk while avoiding two particles occupying the same site. Particles enter a compression state when food is found and this information is propagated in the system, resulting in the system gathering and forming a low-perimeter cluster (compressing) around the food. We prove the following:

▶ **Theorem 1.** *Starting from any valid configuration, in the presence of a single food particle that remains static for a sufficient amount of time, the Adaptive α-Compression algorithm will converge to an α-compressed configuration, for any $\alpha > 1$, connected to the food particle at stationarity with high probability. Conversely, if there are no food particles in the system for a sufficient amount of time, the system converges to a uniform distribution over all possible assignments of particles to sites on the lattice.*

We believe Adaptive α-Compression is the first adaptive algorithm to leverage a *self-induced phase change as an algorithmic tool.* The challenge is to share information locally and autonomously so that eventually most particles enter the correct state and the system exhibits the appropriate phase behavior. We rely on token passing for the system to be able to collectively transition between (multiple, possibly overlapping and interfering) gather and search phases: Each particle locally adjusts its ferromagnetic bias parameter λ to be high when it receives compression tokens, which are continuously generated by any particle in contact with the food source, and to be low when it receives dispersion tokens, which are flooded through the network once a food particle disappears. In order to ensure that, our token passing scheme needs to be carefully engineered so that when the food particle moves or vanishes, the *rate at which the compressed cluster around the food dissipates (via particles returning to the dispersion state) outpaces the rate at which the cluster may continue to grow (via particles joining the cluster in a compression state)*, and thus that the broadcast wave of dispersion tokens will always outpace the broadcast wave of compression tokens, ensuring that whenever we have a situation where two phase change waves compete, the dispersion wave will be the one which wins out in the end. This is done via a novel potential function argument that carefully sets the dispersion and compression token passing probabilities.

We note that while Adaptive α-Compression is very similar to the non-adaptive compression algorithm [1] in the presence of food, allowing particles to compress around a single fixed point (the food particle), this is a nontrivial generalization. Even proving ergodicity of the underlying Markov Chain in the presence of a fixed (food) point from which other particles cannot disconnect is quite complicated and does not follow directly from [1].

References

1 Sarah Cannon, Joshua J. Daymude, Dana Randall, and Andréa W. Richa. A Markov chain algorithm for compression in self-organizing particle systems. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 279–288, 2016.

2 Shengkai Li, Bahnisikha Dutta, Sarah Cannon, Joshua J. Daymude, Ram Avinery, Enes Aydin, Andréa W. Richa, Daniel I. Goldman, and Dana Randall. Programming active granular matter with mechanically induced phase changes. *Science Advances*, 7(17):eabe8494, 2021.

Brief Announcement: Temporal Locality in Online Algorithms

Maciej Pacut
Technische Universität Berlin, Germany

Mahmoud Parham
Faculty of Computer Science, Universität Wien, Austria

Joel Rybicki
Institute of Science and Technology Austria, Klosterneuburg, Austria

Stefan Schmid
TU Berlin, German
Fraunhofer SIT, Berlin, Germany

Jukka Suomela
Aalto University, Espoo, Finland

Aleksandr Tereshchenko
Aalto University, Espoo, Finland

Abstract

Online algorithms make decisions based on past inputs, with the goal of being competitive against an algorithm that sees also future inputs. In this work, we introduce *time-local online algorithms*; these are online algorithms in which the output at any given time is a function of only T latest inputs. Our main observation is that time-local online algorithms are closely connected to *local distributed graph algorithms*: distributed algorithms make decisions based on the *local information in the spatial dimension*, while time-local online algorithms make decisions based on the *local information in the temporal dimension*. We formalize this connection, and show how we can directly use the tools developed to study distributed approximability of graph optimization problems to prove upper and lower bounds on the competitive ratio achieved with time-local online algorithms. Moreover, we show how to use *computational techniques* to synthesize optimal time-local algorithms.

2012 ACM Subject Classification Theory of computation → Online algorithms; Theory of computation → Distributed computing models

Keywords and phrases Online algorithms, distributed algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.52

Related Version *Full Version*: https://arxiv.org/abs/2102.09413

Funding This research has received funding from the German Research Foundation (DFG), grant 470029389 (FlexNets), 2021-2024, and the Marie Skłodowska-Curie grant agreement No. 840605.

1 Introduction

A common setting in theoretical computer science is that there is a sequence of n inputs and we need to produce a sequence of n outputs. In the case of classic centralized algorithms, each output may arbitrarily depend on any part of the input. However, there are two key settings in which outputs are produced based on *partial* inputs (see Figure 1):

- In distributed computing, we can interpret the input sequence as a path formed by n computers; each computer holds a local input and each computer has to produce a local output. In this setting, fast distributed algorithms are also local: if the algorithm stops after $T = O(1)$ communication rounds, then the output of computer number i only depends on the inputs of computers $i - T, \dots, i + T$.

© Maciej Pacut, Mahmoud Parham, Joel Rybicki, Stefan Schmid, Jukka Suomela, and Aleksandr Tereshchenko;
licensed under Creative Commons License CC-BY 4.0

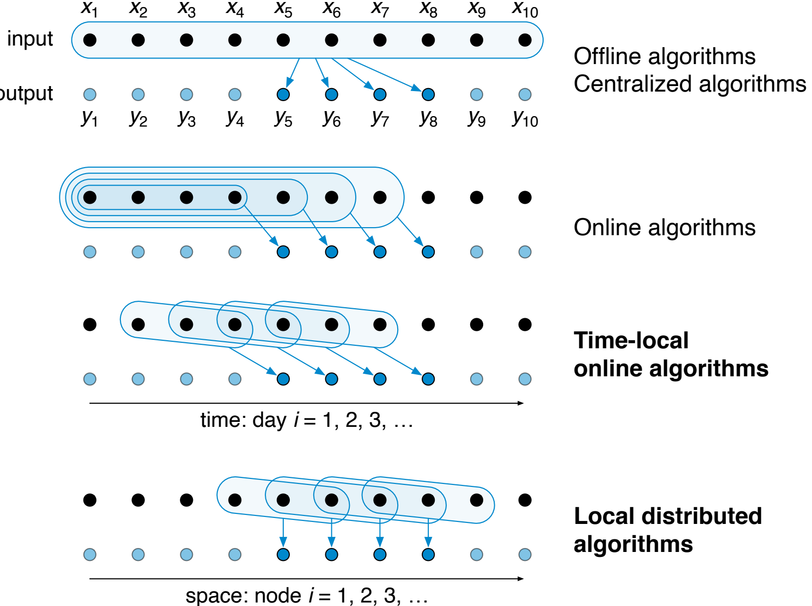


Figure 1 Local decision-making in time vs. space dimensions.

- In online algorithms [3], we can interpret the input sequence as a time series, and the output sequence as a sequence of decisions. At each time point i we need to make a decision that is based on past inputs. That is, output at time point i only depends on the inputs at time points $1, \dots, i-1$.

While these two settings share the feature that each output value depends only on some but not all input values, this connection does not seem to enable much technology transfer between the two domains. In particular, online algorithms are fundamentally infinite objects, as the output may depend on an unbounded number of previous inputs.

In this work, we introduce *time-local online algorithms*; these are online algorithms in which the output at any given time is a function of only T latest inputs, instead of the full history of past inputs. Such algorithms (1) have new attractive properties that are not exhibited by general online algorithms and (2) have many similarities with local distributed graph algorithms, enabling one to transfer tools and techniques between the two domains.

2 Benefits of Time-Local Online Algorithms

Fault-Tolerant Distributed Decision. Time-local online algorithms lead to fault-tolerant distributed decision-making. Consider a setting in which many geographically distributed computers need to make *consistent* decisions. All computers can observe the same input stream, and each day each of them has to announce its own decision.

If all computers are started at the same time, we can take any deterministic online algorithm and let each computer run its own copy of the algorithm. However, this approach does not *tolerate failures*: if a computer crashes and is restarted, the local state of the algorithm is lost, and as the decisions may depend in general on the full history of inputs, it will no longer make consistent decisions with the others.

Deterministic time-local online algorithms automatically guarantee that all computers will make consistent decisions. The system will tolerate an arbitrary number of failures and ensure that the computers will also recover from transient faults, i.e., it is *self-stabilizing* [4]: in T steps since the latest failure, all computers will deterministically make consistent decisions, without any communication.

Random Access to the Decision History. Time-local online algorithms make it possible to efficiently access any past decision with zero additional storage beyond the storage of the input stream. To recover a past decision at any time i, it is sufficient to look up the last T inputs at time i and apply the deterministic time-local algorithm.

3 Connection with Distributed Computing

As illustrated in Figure 1, time-local online algorithms are very similar to local distributed algorithms in directed paths: distributed algorithms make decisions based on the *local information in the spatial dimension*, while time-local online algorithms make decisions based on the *local information in the temporal dimension*. One key difference is that time-local online algorithms are *one-sided* – output i depends only on previous T inputs – while local distributed algorithms are *two-sided* – output i can depend on T inputs in either direction. However, it is easy to navigate between these two settings.

We consider two variants of time-local online algorithms. *Unclocked* algorithms make a decision at time i without knowing the value of i. Such algorithms very similar to deterministic distributed algorithms in the *port-numbering model* [1] – in particular, we face the same challenge of local symmetry breaking. *Clocked* time-local online algorithms can depend on the value of i. Such algorithms turn out to be similar to deterministic distributed algorithms in the *supported* LOCAL model [5]. The key difference is that clocked time-local online algorithms do not know the length of the input sequence in advance, while in the supported LOCAL model the input size is also known.

4 Algorithm Synthesis

In the full version of this work, we describe an *algorithm synthesis method* that one can use to design optimal time-local online algorithms for small values of T, for problems with finite input and output domains. We demonstrate the power of the technique in the context of a variant of the *online file migration problem* [2], and show that e.g. for two nodes and unit migration costs there exists a 3-competitive time-local algorithm with horizon $T = 4$, while no deterministic online algorithm (in the classic sense) can do better.

References

1 Dana Angluin. Local and global properties in networks of processors. In *Proc. 12th Annual ACM Symposium on Theory of Computing (STOC 1980)*, 1980. doi:10.1145/800141.804655.

2 Marcin Bienkowski. Migrating and replicating data in networks. *Computer Science-Research and Development*, 27(3):169–179, 2012.

3 Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

4 Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

5 Klaus-Tycho Foerster, Juho Hirvonen, Jukka Suomela, and Stefan Schmid. On the power of preprocessing in decentralized network optimization. In *Proc. 28th IEEE Conference on Computer Communications (INFOCOM 2019)*, 2019. doi:10.1109/INFOCOM.2019.8737382.