Communication Efficient Self-Stabilizing Leader Election

Xavier Défago
Tokyo Institute of Technology, Japan
defago@c.titech.ac.jp

Yuval Emek
Technion – Israel Institute of Technology, Haifa, Israel
yemek@technion.ac.il

Shay Kutten
Technion – Israel Institute of Technology, Haifa, Israel
kutten@ie.technion.ac.il

Toshimitsu Masuzawa
Osaka University, Japan
masuzawa@ist.osaka-u.ac.jp

Yasumasa Tamura
Tokyo Institute of Technology, Japan
tamura@c.titech.ac.jp

Abstract

This paper presents a randomized self-stabilizing algorithm that elects a leader r in a general n-node undirected graph and constructs a spanning tree T rooted at r. The algorithm works under the synchronous message passing network model, assuming that the nodes know a linear upper bound on n and that each edge has a unique ID known to both its endpoints (or, alternatively, assuming the KT_1 model). The highlight of this algorithm is its superior communication efficiency: It is guaranteed to send a total of $\tilde{O}(n)$ messages, each of constant size, till stabilization, while stabilizing in $\tilde{O}(n)$ rounds, in expectation and with high probability. After stabilization, the algorithm sends at most one constant size message per round while communicating only over the $(n-1)$ edges of T. In all these aspects, the communication overhead of the new algorithm is far smaller than that of the existing (mostly deterministic) self-stabilizing leader election algorithms.

The algorithm is relatively simple and relies mostly on known modules that are common in the fault free leader election literature; these modules are enhanced in various subtle ways in order to assemble them into a communication efficient self-stabilizing algorithm.

2012 ACM Subject Classification Computer systems organization → Fault-tolerant network topologies

Keywords and phrases self-stabilization, leader election, communication overhead

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.11

Related Version A full version that contains all missing proofs as well as additional discussions is available at `http://arxiv.org/abs/2008.04252` [44].

Funding The work of X. Défago, T. Masuzawa, and Y. Tamura was supported by JST SICORP Grant Number JPMJSC1606. The work of Y. Emek and S. Kutten was supported by the Israeli Ministry of Science and Technology (MOST) grant number 3-13565 and by the Technion Hiroshi Fujiwara Cyber Security Research Center and the Israel National Cyber Directorate.
Toshimitsu Masuzawa: The work of T. Masuzawa was supported by JSPS KAKENHI Grant Number 19H04085.

Acknowledgements We are grateful to Valerie King for helpful discussions.

1 Introduction

The *leader election* problem has been recognized early as canonical in capturing the unique characteristics of distributed systems [67, 47, 70, 10]. Together with related problems, such as *spanning tree* construction and *broadcast*, it has been extensively studied through the years in various contexts including mobile networks [71, 81], key distribution [32], routing coordination [78], sensor control [52], general control [51], Paxos and its practical applications [65, 66, 24, 69], peer-to-peer networks [60] and more. The study of efficient algorithms for leader election and its related problems still draws plenty of attention in the present, see, e.g., [59, 76, 72, 48, 45].

A central efficiency measure in this regard is the *communication overhead* that the algorithm adds to the system, in terms of both the number of messages and their size. This has been a topic of interest from the early days, e.g., in [37, 47, 61, 46, 11, 3], to the more recent literature [59, 48, 45], in works of theoretical nature as well as in practically motivated ones [66, 81, 60].

In the realm of *self-stabilizing* algorithms [36] (see [38, 8] for textbooks), communication efficiency is a little bit trickier. Starting from an adversarially chosen initial configuration, the algorithm sends a certain number of messages until it stabilizes; after stabilization, the algorithm is required to keep sending messages indefinitely for the purpose of fault detection (see, e.e., [17]). Consequently, complexity measures related to the algorithm's communication overhead in the self-stabilization literature are divided into two "schools", depending on whether they focus on the messages sent post-stabilization or pre-stabilization.

The motivation behind bounding the communication overhead after the algorithm stabilizes comes from the assumption that faults are relatively rare and most of the time, the system is in a correct configuration. Here, a natural measure is the algorithm's *stabilization bandwidth*, defined in the influential paper of Awerbuch and Varghese [17] as the worst case number of messages sent during any time window of length τ after the algorithm has stabilized, where τ is the algorithm's stabilization time.[1] This measure is justified by the observation that if faults occurred after the algorithm was supposed to have stabilized, then the algorithm must recover from them also in τ time, hence a time window of length τ inherently encapsulates a "full cycle" of the operations executed by the algorithm post-stabilization.

The pre-stabilization approach is motivated by the realization that faulty configurations may lead to bursts of heavy communication, risking an overload of the system's communication components. The natural measure in this regard counts the number of messages sent until the algorithm stabilizes, starting from a worst case initial configuration (see, e.g., [63]).

In this paper, we develop a randomized self-stabilizing algorithm that elects a leader r and constructs a spanning tree T rooted at r in a general n-node (undirected) communication graph, assuming the *synchronous* KT_1 model of [13] (or a slightly weaker version thereof, see Sec. 1.1). Using constant size messages, the algorithm is guaranteed to stabilize in $\tilde{O}(n)$ rounds, while sending $\tilde{O}(n)$ messages, in expectation and whp.[2] [3] The algorithm's stabilization bandwidth is also $\tilde{O}(n)$ as it is guaranteed to send at most one (constant size) message per round after stabilization. Another appealing feature is that after stabilization, the algorithm's communication is restricted to the edges of T.

[1] In [17], this notion is measured per edge, dividing the expression defined in the current paper by the number of edges.

[2] The asymptotic notation $\tilde{O}(x)$ hides polylog(x) factors. Refer to Thm. 2 and 3 for more accurate (asymptotic) bounds.

[3] An event A occurs whp (with high probability) if $\mathbb{P}(A) \geq 1 - n^{-c}$ for an arbitrarily large constant c.

The communication overhead of the new algorithm significantly improves upon the state-of-the-art for self-stabilizing leader election algorithms in general graphs with respect to the aforementioned complexity measures: To the best of our knowledge, no algorithm in the existing literature gets below the $\Omega(m)$ bound for neither the number of messages sent before stabilization, nor the stabilization bandwidth, where m is the number of edges in the graph. In fact, the algorithm's communication efficiency matches (up to logarithmic factors) the state-of-the-art for leader election also in the fault free setting in terms of both the number of messages and the number of bits sent before stabilization.

1.1 Model and Problem

Consider a communication network represented as a simple undirected graph $G = (V, E)$ whose nodes are identified with processing units that may exchange messages of constant size with their neighbors. The execution progresses in synchronous *rounds*, where round $t \in \mathbb{Z}_{\geq 0}$ starts at time t. In each round t, node $v \in V$ (1) receives the messages sent to it over its incident edges in round $t - 1$ (if any); (2) performs local computation; and (3) sends messages over a subset of its incident edges.

Let $n = |V|$. Each edge $e \in E$ admits a unique ID, represented as a bit string of size $O(\log n)$, that is known to both endpoints of e (a slightly weaker assumption than that of the KT_1 model [13]). The nodes also know a linear upper bound N on n that is assumed to be a sufficiently large power of 2.

The goal in the *leader election* problem is to reach a configuration where exactly one node $r \in V$ is marked as a leader (each node $u \in V - \{r\}$ knows that it is not the leader). In this paper, we also require that the nodes maintain a spanning tree of G rooted at the leader r.

We wish to develop a *self-stabilizing* leader election algorithm, where the initial configuration, at time 0, is determined by a malicious adversary that knows the algorithm's code but is oblivious to its random coin tosses (if any). When determining the initial configuration, the adversary may set all variables maintained by node $v \in V$, including its local clock (if such a variable is maintained by v) and incoming messages, with the exception of the variables that store the IDs of v's incident edges and the upper bound N on n.

1.2 Additional Related Work and Discussion

The shortage of work on communication overhead in the context of self-stabilization may have resulted from the fact that any non-trivial self-stabilizing system must send messages infinitely often. This makes its "message complexity" (as defined for non-self-stabilizing systems) infinite [17], giving rise to multiple competing communication measures. Luckily, the current algorithm is efficient in all the suggested measures.

Another possible explanation is that the known self-stabilizing leader election algorithms were not developed from the efficient non-self-stabilizing leader election techniques that preceded them, e.g., they are not derived from the seminal work of Gallager et al. [47]. This is probably because turning a sophisticated algorithm, designed for a fault free environment, into a self-stabilizing one is often a complex task which is prone to mistakes. Hence, less involved, though communication wasteful, methods were the common choice for the design of self-stabilizing algorithms.[4] This contrasts the algorithm developed in the current paper, assembled mainly from modules that are common in the fault free leader election literature.

[4] In explaining how to design an efficient fault tolerant system such as an efficient implementation of Paxos, Lampson writes that "More efficiency means more complicated invariants" and quotes Dijkstra as saying that "An efficient program is an exercise in logical brinkmanship" [66].

Early research on self-stabilization concentrated initially on whether a self-stabilizing algorithm is at all possible for a given task [36, 2, 42, 58, 5] and the required memory size (see, e.g., [36, 5, 39]). The main emphasis later has been on reducing the time complexity in various forms, such as the number of rounds, the asynchronous time, or the number of steps.

In addition to the aforementioned measures for the communication overhead of a self-stabilizing algorithm, it was suggested to measure self-stabilization (and also the related weak detectors) communication efficiency by the number of links over which the local checking is performed indefinitely after stabilization, or the maximum number of such links per node [6, 33, 35, 75, 74, 30]. For leader election, the optimum is $n-1$ links [68, 33]. We note that the algorithm presented in the current paper exactly matches this optimal bound.

Reducing communication overhead is also mentioned as a motivation for reducing the number of steps before stabilization, e.g., in [28], where it is also argued that the number of steps in the shared memory model is closely related to the amount of communication needed to implement that model. The step complexity has been addressed in other papers as well, e.g., [36, 34, 26].

Reducing the communication overhead is also stated as a primary advantage of silent self stabilizing algorithms [39]. In algorithms where neighbors exchange the description of their states periodically (e.g., when using local checking), reducing the state size translates to a reduced message size. Approaches for reducing the message size for detection (though not necessarily the number of messages) by sending only some compressed versions of the state were studied in [41, 77]. The number of messages in certain algorithms with a single starter (irrelevant to the leader election task) is reduced in [43], but is still $\Omega(n^2)$ until stabilization and $\Omega(n)$ per time unit after stabilization, when applied in synchronous networks.

The tasks of leader election and tree constructions are common building blocks in numerous algorithms and practical systems such as mutual exclusion [67], handling various race conditions and topology dependent tasks [47, 24], ensuring that the components in distributed applications remain consistent [64], implementing databases and data centers [55], locks [22], file servers [49, 25], broadcast and multicast [79, 23], and topology update and virtually every global task [12]. In the context of self-stabilization, given a spanning tree, it is possible to apply a self-stabilizing reset, and to use the reset as a module of a transformer that can transform non-self-stabilizing algorithms to be self-stabilizing [9, 17, 15, 16, 5]. The task of converting leader election algorithms themselves to be self-stabilizing has not enjoyed the help of such transformers, since most transformers use a leader and/or a spanning tree.

Numerous self-stabilizing leader election algorithms appeared in the literature; we mention here only a few [4, 9, 17, 42, 14, 40, 1, 53, 21, 19, 31, 7, 62, 20]. Non constant space is needed for leader election [18] by a deterministic protocol even under a centralized daemon if the nodes identities (IDs) are not bounded. Logarithmic space is needed if the algorithm is silent [39].

The algorithm presented in the current paper utilizes a token circulation and timeouts. Self-stabilizing token circulation algorithms in general networks have been treated in numerous papers, e.g., [54, 57, 56, 29, 80]. Multiple papers deal with the related problem of a self-stabilizing construction of a depth first search tree, starting with [27]. Some self-stabilizing tree construction algorithms are implemented on top of tokens (or similar messages that are logically sent to nodes that are not immediate neighbors), e.g., [53, 19]. Timeouts are commonly used in distributed computing, including in leader election algorithms, e.g., [46, 53, 19].

The KT_1 model is advocated in [13] as being more realistic than KT_0, where a node only knows its ports, but not the node connected to each port. In recent years, several non-self-stabilizing algorithms were proposed to explore the properties of KT_1 in order to reduce the message complexity [59, 48, 50, 72, 73].

1.3 Paper's Organization

The rest of the paper is organized as follows. Following some notation and terminology defined in Sec. 2, the algorithm is described in Sec. 3, starting with an informal overview (Sec. 3.1) that includes a discussion of the main technical ideas. Sec. 4 presents a sketch of the algorithm's analysis, including its fault recovery guarantees, stabilization run-time, and message complexity; refer to [44] for the complete analysis.

2 Preliminaries

Throughout this paper, it is assumed that (directed and undirected) graphs may include self loops, but not parallel edges. We denote the node set and edge set of a (directed or undirected) graph G by $\mathcal{V}(G)$ and $\mathcal{E}(G)$, respectively.

Consider an undirected graph G. A subgraph T of G that admits a tree topology is called a *subtree* of G. Two node disjoint subtrees T and T' of G are said to be *adjacent* if there exists an edge $e = \{v, v'\} \in \mathcal{E}(G)$ such that $v \in \mathcal{V}(T)$ and $v' \in \mathcal{V}(T')$; such an edge e is referred to as a *crossing* edge. A *merger* of the subtrees T and T' (over the crossing edge e) forms a new subtree of G whose node set is $\mathcal{V}(T) \cup \mathcal{V}(T')$ and whose edge set is $\mathcal{E}(T) \cup \mathcal{E}(T') \cup \{e\}$.

Consider a directed graph D. The *undirected version* of D is the undirected graph obtained from D by ignoring the edge directions. We say that D is *weakly connected* if the undirected version of D is connected (note the distinction from the notion of a strongly connected directed graph).

The directed graph D is said to be a *pseudoforest* if the outdegree of every node $v \in \mathcal{V}(D)$ is at most 1. A *pseudotree* is a weakly connected pseudoforest. The undirected versions of a pseudoforest and a pseudotree are referred to as an *undirected pseudoforest* and an *undirected pseudotree*, respectively.

We subsequently reserve the notation G for the n-node undirected communication graph and denote $V = \mathcal{V}(G)$ and $E = \mathcal{E}(G)$. For a node $v \in V$, its neighbor set in G is denoted by $N(v) = \{u \in V \mid \{u, v\} \in E\}$.

3 The Algorithm

3.1 An Informal Overview

In this section, we provide an overview of our algorithm, composed of various modules that the informed reader will recognize from the vast literature on leader election and spanning tree construction in fault free environments. The algorithm maintains a partition of the nodes into rooted trees, defined by means of variables $v.\texttt{prnt}$ and $v.\texttt{chld}$ in which each node v stores its tree parent and children, respectively. The actions of each tree T are coordinated by its root, namely, the unique node $r \in \mathcal{V}(T)$ satisfying $r.\texttt{prnt} = \perp$. The goal is to repeatedly merge the trees over crossing edges until a single tree that spans the whole graph remains.

The algorithm's main module, denoted by `Main`, divides the execution into *phases*, where in each phase the root r of tree T decides at random between two modules, denoted by `Propose` and `Accept`, to be invoked in the current phase. The role of `Propose` is to find a crossing edge over which a merger proposal is sent. To this end, r invokes a randomized module, denoted by `Cross`, that implements procedure *FindAny* of [59]. If `Cross` fails to find a crossing edge, then the phase ends.

Otherwise, an edge $\{x, y\}$, that crosses between $\mathcal{V}(T) \ni x$ and $V - \mathcal{V}(T) \ni y$, is returned. Then, r invokes a module, denoted by `Trnsfer`, whose role is to update the `prnt` and `chld` variables along the unique simple (r, x)-path in T so that T is re-rooted at x (cf. [47]). Following that, x sends a merger proposal to y and waits silently for a reply. If such a reply does not arrive within the next $\Theta(N)$ rounds, then the phase ends and x, now being the root, initiates a new phase. Otherwise, x becomes the child of y, merging T into y's tree.

The role of `Accept` is to accept incoming merger proposals. In particular, a merger proposal sent from a node x to a node y in tree T' is accepted by y, sending a reply to x, if and only if the current phase of T' ("as far as y knows") is an `Accept` phase. This ensures that the resulting structure is cycle free.

The aforementioned process is implemented on top of a module, denoted by `Traverse`, that implements a depth first search traversal of the tree by a conceptual *token.* The executions of both `Cross` and `Accept` are divided into $\Theta(\log N)$ *epochs* so that `Traverse` is invoked by the root r at the beginning of each epoch. The epochs lasts for a fixed $\Theta(N)$ number of rounds, chosen to guarantee that the traversal can be safely completed, returning the token to r where it is stored until the next epoch begins. Apart from `Traverse` that controls the token's mobility, `Trnsfer` also shifts the token down the root transfer path, thus ensuring that outside the scope of the traversals handled by `Traverse`, the token is always stored at the root.

A key property of the algorithm, that facilitates its communication efficiency, is that a node may send a message only when it holds the token (and then, it may send at most one message per round). In particular, `Cross` implements procedure *FindAny* [59] on top of the tree traversals, so that each one of its $\Theta(\log N)$ epochs is responsible for one broadcast-echo process in the tree (carrying $O(1)$ bits of information). Moreover, a merger proposal sent from node x to node y is recorded at y (assuming that y's tree is in an `Accept` phase) until y holds the token as part of a traversal of its tree; the proposal is then processed and y sends a reply to x. The token held by x prior to the merger is dissolved when x's tree is merged into y's, striving for the invariant that each tree holds a single token.

The fact that each tree is repeatedly traversed by its token is exploited by the algorithm's fault detection mechanism: Each node v maintains a $v.$`tmr` variable that is reset when v receives the token from its parent as part of a traversal and is incremented in every round otherwise; if $v.$`tmr` exceeds a predetermined $\Theta(N)$ threshold, then v invokes a procedure named `Restart` that resets all its variables, so that v forms a new singleton tree, and generates a fresh token at v. A malformed tree is (implicitly) detected by the token being lost when it is sent to v from an adjacent node u while v does not "expect" to receive a token from u. We emphasize that in both events (v experiencing a restart and v ignoring u's message), v does not inform any of its neighbors of the detected fault.

This fault detection mechanism essentially replaces the local checking module, common to many self-stabilizing algorithms: rather than checking "all neighbors all the time", node v checks only the tokens it receives, which is clearly more efficient communication-wise. Our algorithm's self-stabilization guarantees follow from this mechanism with some additional fine points explained in Sec. 3.2.

3.2 A Detailed Description

Our algorithm performs leader election by constructing a rooted spanning tree of G. To this end, each node $v \in V$ maintains a variable $v.$`prnt` $\in N(v) \cup \{\perp\}$ that points to its tree parent and a variable $v.$`chld` $\in (N(v))^*$ that stores a list of pointers to its tree children; by a slight abuse of notation, we may address $v.$`chld` as a subset of $N(v)$ when the order of its elements is not important.

We say that v is a *root* if v.prnt $= \perp$. Let v.T_nbrs $= v$.chld if v is a root; and v.T_nbrs $= v$.chld $\cup \{v$.prnt$\}$ if v is not a root. We refer to the nodes in v.T_nbrs as the *tree neighbors* of v (from the perspective of v).

For the sake of simplifying the algorithm's description, we augment the communication graph with a virtual *shadow* node $\tilde{v}$ for each node v of the original graph so that $\tilde{v}$ has a single incident edge connecting it to v. The algorithm is designed so that v is always the parent of $\tilde{v}$ which guarantees, in particular, that the set of tree neighbors of a node is never empty. To distinguish the original graph nodes from their virtual shadows we refer to the former as *physical* nodes. The operation of the virtual shadow node $\tilde{v}$ is simulated by its corresponding physical node v; this simulation is straightforward as v is the only neighbor of $\tilde{v}$. We assume hereafter that the node set V of the communication graph $G = (V, E)$ contains both the physical and shadow nodes.

The aforementioned modules Traverse, Main Propose, Accept, Cross, and Trnsfer are presented in Sec. 3.2.1–3.2.6, respectively, while Restart is presented in Sec. 3.2.7. To help the reader put things into context, some parts of the algorithm's description are written as if no "recent faults" has occurred; we emphasize that the correctness of the algorithm does not rely on any such assumption. Moreover, to clarify the algorithm's presentation, we restrict our attention to the less trivial components, leaving out some details that the reader can easily complete by themselves; for ease of reference, Table 1 summarizes the variables maintained by these components.

Table 1 The variables of node $v \in V$.

Variable	Range	Semantics
prnt	$N(v) \cup \{\perp\}$	v's parent
chld	$(N(v))^*$	v's children
T_nbrs	$2^{N(v)}$	v's tree neighbors
tkn	$\{0, 1\}$	indicates that v holds a token
tkn_d	T_nbrs	the direction of the token
recent	$\{0, 1\}$	indicates that v has passed a token in the previous round
tmr	$\mathbb{Z}_{\geq 0}$	#rounds since receiving a (hot) token
out_prop	$N(v) \cup \{\perp\}$	the direction of the (recently found) crossing edge
in_prop(u)	$\{0, 1\}$	indicates that a proposal from $u \in N(v)$ has been registered

3.2.1 Module Traverse

Consider some root node r and let T be r's tree. The Traverse module implements a distributed process in which a conceptual *token* is passed from node to node, using designated pass_tkn messages, to form a depth first search traversal of T.

To monitor the token distribution, each node $v \in V$ maintains three variables: (a) v.tkn $\in \{0, 1\}$ that indicates whether v holds a token; (b) v.tkn_d $\in v$.T_nbrs that points to the last tree neighbor to which v has passed a token; and (c) v.recent $\in \{0, 1\}$ that indicates whether v has passed a token in the previous round.

Denoting $d = |v$.T_nbrs$|$, node v also employs a permutation (i.e., a bijection) π_v : $\{0, 1, \ldots, d-1\} \to v$.T_nbrs defined so that $\pi_v(i-1)$ points to the i-th element in v.chld for $1 \leq i \leq |v$.chld$|$ and $\pi_v(d-1)$ points to v.prnt if v.prnt $\neq \perp$. This permutation naturally induces a periodic sequence over the nodes in v.T_nbrs so that the i-th element of this sequence is $\pi_v(i \bmod d)$ for every $i \in \mathbb{Z}_{>0}$, referring to $\pi_v(i + 1 \bmod d)$ as its π_v*-successor*. If v is a physical node, which ensures that v.chld $\neq \emptyset$, then we refer to $\pi_v(0)$ and to $\pi_v(|v$.chld$| - 1)$ as v's *first child* and *last child*, respectively.

Consider some non-root node v. Upon receiving a `pass_tkn` message M from node $u \in N(v)$, node v verifies that (i) $v.\texttt{tkn} = 0$; (ii) $v.\texttt{recent} = 0$; (iii) $v.\texttt{tkn\_d} = u$; and (iv) its parent-child relations with u are consistent between the two nodes, that is, either $u = v.\texttt{prnt}$ and M indicates that u is the parent of v or $u \in v.\texttt{chld}$ and M indicates that u is a child of v. If any of these four conditions is not satisfied, then v ignores M altogether, thus causing the token it carries to (conceptually) disappear.

Assuming that the four conditions are satisfied, node v sets $v.\texttt{tkn} \leftarrow 1$, thus indicating that it now holds the token that conceptually arrived with message M. Following that, v holds the token for 1–4 full rounds, where the exact number is determined by the modules implemented on top of `Traverse`. Then, v passes the token to the π_v-successor u' of $u = v.\texttt{tkn\_d}$ by sending to it a `pass_tkn` message and sets (1) $v.\texttt{tkn} \leftarrow 0$; (2) $v.\texttt{tkn\_d} \leftarrow u'$; and (3) $v.\texttt{recent} \leftarrow 1$. The sole purpose of the flag $v.\texttt{recent}$ is to ensure that v holds no token for (at least) one full round before it can accept a token again. This flag is always turned off one round after it has been turned on; that is, v resets $v.\texttt{recent} \leftarrow 0$ in each round, after the value of $v.\texttt{recent}$ has been read as part of processing the incoming messages.

The operation of the root r (which is always a physical node) in `Traverse` is identical to that of any non-root node except that r is also responsible for initiating the traversal, by passing the token to its first child, and terminating the traversal, after receiving the token from its last child, holding the token in between traversals. To distinguish the situation where the token is held by a (root or non-root) node in the midst of a traversal from the situation in which the token is held by a (root) node between traversals, we refer to the token in the former situation as *hot* and in the latter situation as *cold*. The traversal process is initiated at r upon receiving a designated signal from the modules implemented on top of `Traverse`. If r does not hold a cold token when this signal is received, then `Restart` is invoked; otherwise, the token becomes hot and a traversal of T is initiated.

Consider some traversal of T. We refer to the `pass_tkn` message received (resp., sent) by a non-root node v from (resp., to) its parent as v's *discovery message* (resp., *retraction message*) and to the round in which this message is received (resp., sent) as v's *discovery round* (resp., *retraction round*). The *discovery round* (resp., *retraction round*) of T's root r is defined to be the round in which the traversal is initiated (resp., terminated), i.e., the round in which the token held by r turns from cold to hot (resp., from hot to cold). A key observation is that the traversal allows r to simulate a broadcast-echo process over its tree T by piggybacking the content of the broadcast (resp., echo) messages over the discovery (resp., retraction) messages.

`Traverse` is responsible also for a timer mechanism monitored by variable $v.\texttt{tmr} \in \mathbb{Z}_{\geq 0}$ that each node $v \in V$ maintains. This variable is incremented by v in every round. During a discovery round, v verifies that $v.\texttt{tmr} \geq C_{\text{tr}} N$, where C_{tr} is a positive constant whose value is determined in Sec. 4. If this condition is not satisfied, an event referred to as a *premature discovery*, then v invokes `Restart`; otherwise, v resets $v.\texttt{tmr} \leftarrow 0$. If, at any stage of the execution, variable $v.\texttt{tmr}$ exceeds the $8 C_{\text{tr}} N$ threshold, then v also invokes `Restart`; this event is referred to as an *expiration* of $v.\texttt{tmr}$.

3.2.2 Module `Main`

The execution of `Main` is partitioned into *phases* that are further partitioned into *epochs*; these partitions are orchestrated by the root nodes r. Each phase is dedicated to an invocation of either `Propose` or `Accept`; the root r decides between the two at the beginning of the phase by tossing a fair coin.

A `Propose` phase consists of $C_{\mathrm{ph}} \log N$ *search* epochs, where $C_{\mathrm{ph}} \in \mathbb{Z}_{>0}$ is a constant to be determined in the sequel, followed by a *root transfer* epoch, followed by a *proposing* epoch. The search epochs are dedicated to an invocation of `Cross` and based on its outcome, the root transfer epoch is either empty, in which case, the proposing epoch is also empty, or dedicated to an invocation of `Trnsfer`. An `Accept` phase consists of $C_{\mathrm{ph}} \log N + 2$ *accepting* epochs.

Consider a tree T rooted at r. Each search and accepting epoch is dedicated to an invocation of `Traverse`, initiated by r, during which the (hot) token completes one traversal of T and returns to r, where it is held (cold) until the epoch ends. These epochs have a fixed length of $2C_{\mathrm{tr}}N$ rounds. The length of the root transfer and proposing epochs may vary, however, it is guaranteed that their combined length is at most $4C_{\mathrm{tr}}N$ rounds. Therefore, the `Accept` phases last for exactly $(C_{\mathrm{ph}} \log N + 2) \cdot 2C_{\mathrm{tr}}N$ rounds, whereas the length of each `Propose` phase is at least $C_{\mathrm{ph}} \log N \cdot 2C_{\mathrm{tr}}N$ and at most $(C_{\mathrm{ph}} \log N + 2) \cdot 2C_{\mathrm{tr}}N$ rounds.

A variable that plays an important role in the distinction between `Accept` and `Propose` phases is v.`accpt` $\in \{0, 1\}$, maintained by each node $v \in \mathcal{V}(T)$, that indicates weather v is available for merger proposals (more on that soon). Node v turns this variable on (if it is not on already) in the retraction round of any traversal associated with an accepting epoch (i.e., during `Accept` phases); it turns this variable off (if it is not off already) in the discovery round of any traversal associated with a search epoch (i.e., during `Propose` phases). This means that the `accpt` variables are turned on in a bottom up order and turned off in a top down order. Notice that once the variable v.`accpt` is on (resp., off), it stays on (resp., off) at least until the end of the current `Accept` (resp., `Propose`) phase.

3.2.3 Module `Propose`

Consider some tree T rooted at node r. `Propose` controls the algorithm's operation during `Propose` phases orchestrated by r. Specifically, during the $C_{\mathrm{ph}} \log N$ search epochs of a `Propose` phase, r orchestrates the search for an edge that crosses from T to an (arbitrary) adjacent tree. This is done by invoking `Cross` that is guaranteed to return a crossing edge with a positive constant probability if such an edge exists.

The output of `Cross` is returned by means of the variables v.`out_prop` $\in N(v) \cup \{\perp\}$ that each node $v \in \mathcal{V}(T)$ maintains. Assuming that no faults have occurred during the execution of `Cross`, it is guaranteed that these variables satisfy the following three properties: (OP1) If r.`out_prop` $= \perp$, then v.`out_prop` $= \perp$ for every node $v \in \mathcal{V}(T)$. (OP2) If r.`out_prop` $\in N(v)$, then there exists exactly one *crossing port* $x \in \mathcal{V}(T)$, namely, a node satisfying x.`out_prop` $\in N(x) - x$.`T_nbrs`. Taking P to be the unique simple (r, x)-path in T, the variable v.`out_prop` points to the successor of v along P for every node $v \in \mathcal{V}(P) - \{x\}$; and v.`out_prop` $= \perp$ for every node $v \in \mathcal{V}(T) - \mathcal{V}(P)$. (OP3) If x is a crossing port, then x.`out_prop` $= y \in V - \mathcal{V}(T)$. Refer to Figure 1 for an illustration.

If `Cross` does not find a crossing edge, indicated by r.`out_prop` $= \perp$, then the phase terminates (which means that the root transfer and proposing epochs are empty). Assuming that `Cross` returns the edge $\{x, y\}$ that connects the crossing port $x \in \mathcal{V}(T)$ with a node $y \in V - \mathcal{V}(T)$, the root transfer epoch is dedicated to modifying the parent-child relations in T so that the root role transfers from r to x; this is handled by `Trnsfer`.

When `Trnsfer` terminates, the tree T is rooted at x, that also holds the (cold) token, and x.`out_prop` $= y$. The proposing epoch is now dedicated to the attempt of x to merge T and y's tree T' over the crossing edge $e = \{x, y\}$. To this end, in the first round of the proposing epoch, x sends a `propose` message to y. Following that, x waits silently for a reply from y in the form of an `accept` message. If this reply does not arrive during the subsequent

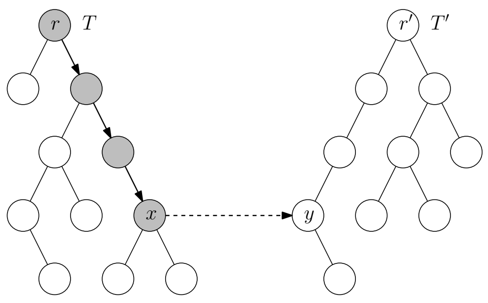

Figure 1 Towards a merger of the proposing tree T rooted at r and the accepting tree T' rooted at r' over the crossing edge $\{x, y\}$. The tree edges are depicted by the solid lines whereas the crossing edge is depicted by the dashed line. The nodes along the unique (r, x)-path in T are marked in gray and the values of their `out_prop` variables are depicted by the oriented edges.

$3C_{\mathrm{tr}}N - 1$ rounds, then x sets x.`out_prop` $\leftarrow \perp$ and the proposing epoch terminates (with no tree merger), also terminating this `Propose` phase (the next phase starts with x as the root of T).

If x receives an `accept` message from y before the end of the proposing epoch, then T is merged with T'. From the perspective of x, this merger is executed by setting (1) x.`prnt` $\leftarrow y$; (2) x.`tkn` $\leftarrow 0$; (3) x.`tkn_d` $\leftarrow y$; and (4) x.`out_prop` $\leftarrow \perp$. In particular, this means that x is no longer a root and that it expects the next `pass_tkn` message to arrive from its new parent y. Moreover, the token held by x is eliminated, an event referred to hereafter as *dissolving* x's token.

3.2.4 Module `Accept`

Consider some tree T' rooted at r'. `Accept` controls the algorithm's operation during `Accept` phases orchestrated by r'. The role of an `Accept` phase is to allow the nodes of T' to accept merger proposals of neighboring nodes in adjacent trees. To this end, each node $y \in \mathcal{V}(T')$ maintains the binary variable y.`in_prop`$(x) \in \{0, 1\}$ for every (graph) neighbor $x \in N(y) - y$.`T_nbrs`. This variable registers merger proposals received from x so that they can be processed when y is ready (as explained soon). Specifically, y.`in_prop`(x) is turned on when y receives a `propose` message while y.`accpt` $= 1$; it is turned off when y resets y.`accpt` $\leftarrow 0$. We emphasize that any `propose` message received while y.`accpt` $= 0$ is ignored. Moreover, if a node $x \in N(y)$ joins y.`T_nbrs`, then the y.`in_prop`(x) entry in y.`in_prop`$(\cdot)$ is eliminated.

An `Accept` phase consists of $C_{\mathrm{ph}} \log N + 2$ accepting epochs, each one of them begins by signaling r' to invoke `Traverse`, thus initiating a traversal process during which a (hot) token is passed through the nodes of T'; when the traversal terminates, the token is held (cold) at r' until the next epoch. A physical node $y \in \mathcal{V}(T')$ is said to be *retraction ready* if it holds a hot token while y.`tkn_d` $= \pi_y(|y.\texttt{chld}| - 1)$, indicating that the held token has been received from y's last child. The algorithm is designed so that y may respond to merger proposals of its neighbors only when it is retraction ready. Notice that a shadow node $\tilde{y}$ never receives merger proposals as its only neighbor y in G is always a tree neighbor of $\tilde{y}$.

Consider node y when it becomes retraction ready while y.`accpt` $= 1$ and let w be the (current) last child of y. Node y checks the content of $P_y = \{x \mid y.\texttt{in\_prop}(x) = 1\}$; if $P_y = \emptyset$, then the role of y in the current accepting epoch is over and `Accept` signals `Traverse` that y can release the hot token (after holding it for one full round), which leads to y's retraction round.

If $P_y \neq \emptyset$, then y picks one neighbor $x \in P_y$ (arbitrarily) and sends to it an `accept` message. Then, in the subsequent round, the merger of T' and the tree T of x is executed, where on the T' side, y adds x as its last child in T' by appending x to the end of y.`chld`, turning it into the π_y-successor of (the original last child) w. Following that, `Accept` signals `Traverse` that y can release the hot token, continuing with the traversal of the updated T'; this results in passing the hot token to x as x is now the π_y-successor of $w = y$.`tkn_d`.

Observe that the tree merger is designed so that T is traversed by the token held by y immediately after it is merged into T'. The aforementioned mechanism that controls the `accpt` variables guarantees that v.`accpt` is turned on during this traversal for every node $v \in \mathcal{V}(T)$; this happens during v's retraction round which means that v will be able to process incoming merger proposals only during the next accepting epoch (if any). Node y continues processing merger proposals of other nodes when it gets the token back from x.

3.2.5 Module `Cross`

King et al. [59] developed a distributed procedure called *FindAny*, working in a fault free environment, that given a subtree T of G rooted at r, finds an edge that crosses between $\mathcal{V}(T)$ and $V - \mathcal{V}(T)$ with a positive constant probability if such an edge exists. The procedure is orchestrated by r using a constant number of broadcast-echo iterations over T with messages of size $O(\log n)$.[5]

We use a variant of the procedure of King et al., where each broadcast-echo iteration with $O(\log n)$ size messages is subdivided into $O(\log n)$ broadcast-echo iterations with constant size messages. `Cross` is then implemented using $O(\log n)$ iterations of the traversal process managed by `Traverse`, piggybacking the (constant size) messages of each broadcast-echo iteration over the `pass_tkn` messages of the corresponding traversal process (see Sec. 3.2.1). Each one of these $O(\log n)$ traversal iterations is executed in its own search epoch, where r is signaled to invoke `Traverse` at the beginning of the epoch. The value of C_{ph} is derived from the hidden constant in the O-notation.

Unfortunately, we cannot guarantee that properties (OP1)–(OP3) of the `out_prop` variables in T (see Sec. 3.2.3) hold if these variables sprout from the adversarially chosen initial configuration, rather than being generated by a complete invocation of `Cross` as part of the execution. However, we implement the module so that property (OP2) is guaranteed as long as the last search epoch, referred to hereafter as the *safety* epoch of `Cross`, is completed as part of the execution.

To this end, we implement `Cross` so that variable v.`out_prop` is set during v's retraction round of the traversal associated with the safety epoch for every node $v \in \mathcal{V}(T)$. Moreover, the retraction message of each child $u \in v$.`chld` of v specifies whether the number of crossing ports in u's subtree is zero, one, or at least two; node v then sets v.`out_prop` $\leftarrow u$ (in its own retraction round) if and only if u admits one crossing port in its subtree and every other child of v admits zero crossing ports in its subtree.

We emphasize that a fault free execution of the safety epoch alone does not guarantee properties (OP1) and (OP3); that is, it does not rule out a scenario where r.`out_prop` $= \perp$ and yet, v.`out_prop` $\neq \perp$ for some non-root nodes $v \in \mathcal{V}(T)$, nor does it rule out a scenario where r.`out_prop` $\neq \perp$, but the returned edge e connects the (unique) crossing port x to another node in T (both scenarios require the adversary's "involvement"). We show in the sequel that the latter scenario does not affect the correctness of our algorithm; to overcome

[5] The procedure needed for our purposes is a slightly simplified version of the one developed in [59].

the former scenario, every node v resets v.out_prop $\leftarrow \perp$ upon receiving a traversal discovery message of Traverse, that is, when v receives a pass_tkn message from v.prnt (and does not ignore it). This is consistent with the fact that Trnsfer is implemented using root_trns messages, rather than pass_tkn messages.

3.2.6 Module Trnsfer

Given a tree T rooted at r and a crossing port x, Trnsfer modifies the parent-child relations in T so that it is re-rooted at x. When the module is invoked, the out_prop variables maintained by the nodes in T mark the unique simple (r, x)-path P in T (assuming that these variables were generated during the safety epoch of Cross, completed as part of the execution). Trnsfer sequentially shifts the root role from node to node down the path P, using root_trns messages that carry a cold token. This is done as follows.

Consider some intermediate node $v \in \mathcal{V}(P)$, as indicated by v.prnt $\neq \perp$ and v.out_prop $\in v$.chld. When v receives a root_trns message M from node $u \in N(v)$, it verifies that (i) v.tkn $= 0$; (ii) v.recent $= 0$; (iii) v.tkn_d $= u$; and (iv) $u = v$.prnt; if any of these four conditions is not satisfied, then v ignores M altogether, thus causing the token that M carries to (conceptually) disappear.

Assuming that the four conditions are satisfied, node v sets v.tkn $\leftarrow 1$, indicating that it now holds the (cold) token, and turns itself into the new root by setting v.prnt $\leftarrow \perp$ and appending u to v.chld. (Note that v does not reset variable v.tmr upon receiving a cold token.) In the subsequent round, v sends a root_trns message to its successor $w = v$.out_prop in P and sets (1) v.tkn $\leftarrow 0$; (2) v.tkn_d $\leftarrow w$; (3) v.out_prop $\leftarrow \perp$; and (4) v.recent $\leftarrow 1$; thus indicating that v no longer holds the token and that it expects the next pass_tkn message to arrive from w. It also turns itself into a child of w by setting v.prnt $\leftarrow w$ and v.chld $\leftarrow v$.chld $- \{w\}$; to ensure that the parent-child relations between v and w are updated in synchrony, these two operations are performed by v in the round subsequent to sending the root_trns message.

The original root r behaves just like an intermediate node except that its actions are triggered by the invocation of Trnsfer, rather than by receiving a root_trns message from its (non-existent) predecessor in P. The crossing port x also behaves just like an intermediate node except that it remains the (cold token holding) root and does not send a root_trns message to its (non-existent) successor in P; rather, it initiates a proposing epoch as explained earlier.

3.2.7 Procedure Restart

When Restart is invoked at a physical node $v \in V$ or at its shadow $\tilde{v}$, we say that both v and $\tilde{v}$ *experience* a restart. Upon invocation of the procedure at either of the two nodes, Restart disconnects v from all its tree neighbors other than $\tilde{v}$ and places v and $\tilde{v}$ in a new tree rooted at v that includes only these two nodes. This is implemented by setting v.prnt $\leftarrow \perp$, $\tilde{v}$.prnt $\leftarrow v$, v.chld $\leftarrow \{\tilde{v}\}$, and $\tilde{v}$.chld $\leftarrow \emptyset$. Restart assigns the new tree's (cold) token to v, setting v.tkn $\leftarrow 1$, $\tilde{v}$.tkn $\leftarrow 0$, and v.recent, $\tilde{v}$.recent $\leftarrow 0$, and consistently adjusts the tkn_d variables by setting v.tkn_d $\leftarrow \tilde{v}$ and $\tilde{v}$.tkn_d $\leftarrow v$. The out_prop and in_prop variables of v and $\tilde{v}$ are initialized, setting v.out_prop, $\tilde{v}$.out_prop $\leftarrow \perp$ and v.in_prop$(u) \leftarrow 0$ for every $u \in N(v) - \{\tilde{v}\}$ (recall that v is the only graph neighbor of $\tilde{v}$ and is also a tree neighbor of $\tilde{v}$, so $\tilde{v}$.in_prop$(\cdot)$ is "empty"), thus indicating that v and $\tilde{v}$ do not participate in any active root transfer path and that no tree merger proposals are currently registered at them. Finally, the tmr variables are set to v.tmr, $\tilde{v}$.tmr $\leftarrow C_{\mathrm{tr}}N$, thus allowing the nodes to start a traversal (without experiencing another restart due to a premature discovery), and v initiates a new Propose phase.

On top of the fault detection mentioned earlier, `Restart` is also invoked whenever the node's internal variables are inconsistent with each other. This can happen only in the initial configuration.

4 Analysis

Notation and Terminology. We analyze the operation of the algorithm on the communication graph $G = (V, E)$ that contains both the physical nodes and their (virtual) shadow nodes (see Sec. 3). Recalling that a node can send a message only when it holds a token, the algorithm's proof of correctness relies on analyzing the token distribution across G and its dynamics over time. To this end, we regard the conceptual tokens as actual entities that are transitioned across the graph.

Throughout this section, we append a superscript t to the notation of our variables when referring to the value that these variables hold at time $t \in \mathbb{Z}_{\geq 0}$ so $v.\mathtt{dummy}^t$ denotes the value held by variable `dummy` of node $v \in V$ at time t. The superscript t is omitted when t is not important or clear from the context.

Consider a node $v \in V$ and suppose that $v.\mathtt{tkn}^t = 1$ which means that v holds a token κ at time t. This token may be *dispatched* in round t from v to a node $w \in v.\mathtt{T\_nbrs}$ by means of a `pass_tkn` message if κ is hot; or a `root_trns` message if κ is cold. We say that the dispatch *fails* and that κ *dies* if this message is ignored by w (in round $t + 1$); otherwise, we say that the dispatch *succeeds* and that κ is *acquired* by w. Notice that in the latter case, we regard the dispatch as successful, saying that w acquires κ, even if w experiences a restart in round $t + 1$ ("after" acquiring the token).

Node $v \in V$ is said to *own* a token at time $t > 0$ if either (1) $v.\mathtt{tkn}^t = 1$; or (2) $v.\mathtt{tkn}^t = 0$ and $v.\mathtt{recent}^t = 1$. In other words, v owns a token at time t if it holds a token at time t or if it has just dispatched a token to a tree neighbor u in round $t - 1$ (which means that v no longer holds it at the beginning of round t). Using this convention, we ensure that if the dispatch of the token from v to u is successful, then the token is delivered smoothly from v to u so that it is owned contiguously by exactly one of the two nodes (avoiding an ownership gap at time t). We say that a token κ is *alive* at time $t > 0$ if it is owned by some node at that time.

When a physical node v and its shadow node $\tilde{v}$ experience a restart, a new token κ is generated at v (and immediately dispatched to $\tilde{v}$). We emphasize that if v or $\tilde{v}$ hold a token κ' in round t prior to the invocation of `Restart`, then we think of κ' as disappearing and regard κ as a new token; this is another case where we say that κ' *dies* in round t.

Edge $e = \{u, v\} \in E$ is said to be *compatible* at time t if $u = v.\mathtt{prnt}^t$ and $v \in u.\mathtt{chld}^t$ (or vice versa). Let $G_c^t = (V, E_c^t)$ be the undirected graph defined by setting $E_c^t \subseteq E$ to be the subset of edges compatible at time t. The (maximal) connected components of G_c^t are referred to as *compatible components*. By definition, G_c^t is an undirected pseudoforest and the compatible components are undirected pseudotrees.

Edge $e = \{v, w\} \in E$ is said to be a *dangling* edge of node v at time t if v regards it as a tree edge, i.e., $w \in v.\mathtt{T\_nbrs}^t$, and yet $e \notin E_c^t$. Notice that e may be a dangling edge of v if w does not regard it as a tree edge or if w does regard it as a tree edge, but there is an inconsistency in the parent-child relations between v and w. We say that e is *dangling* without mentioning v if v is not important or clear from the context.

Fix some node $v \in V$. We make an extensive use of the variable $v.\delta \in v.\texttt{T\_nbrs} \cup \{v\} \cup \{\bot\}$ defined for the sake of the analysis by setting

$$v.\delta^t = \begin{cases} v, & \text{if } v \text{ owns a token at time } t \\ v.\texttt{tkn\_d}^t, & \text{if } v \text{ does not own a token at time } t \text{ and } \{v, v.\texttt{tkn\_d}^t\} \in E_c^t \\ \bot, & \text{otherwise} \end{cases} .$$

Let $G_\delta^t = (V, E_\delta^t)$ be the directed graph defined by setting $E_\delta^t = \{(v, v.\delta) \mid v \in V \wedge v.\delta \neq \bot\}$. The weakly connected components of G_δ^t are referred to as *δ-components.* By definition, G_δ^t is a pseudoforest and the δ-components are pseudotrees. Moreover, all edges of the undirected version of G_δ^t that are not self loops are also edges of G_c^t.

A δ-component is said to be *selfish* if it includes a self loop. A non-selfish δ-component may be cycle free or include a cycle that involves two or more nodes. By definition, node $v \in V$ is incident to a self loop in G_δ^t if and only if it owns a token at time t, which means that the nodes of a selfish δ-component own exactly one token (all together), whereas the nodes of a non-selfish δ-component do not own any token.

For a token κ that is alive at time t, let D_κ^t be the selfish δ-component that includes the node that owns κ and let $\overline{D}_\kappa^t$ be the undirected version of D_κ^t excluding the self loop (notice that $\overline{D}_\kappa^t$ is always a tree). We say that a node $v \in V$ is *covered* by κ at time t if $v \in \mathcal{V}(D_\kappa^t)$; v is said to be *uncovered* at time t if it belongs to a non-selfish δ-component in G_δ^t. A token κ is said to be *strong* at time t if (1) κ is alive at time t; (2) the nodes of D_κ^t admit no dangling edges at time t; and (3) $\overline{D}_\kappa^t$ forms a compatible component in G_c^t.

The selfish δ-components obey the following dynamics. Consider a token κ that is dispatched by a node $v \in V$ in round $t - 1 > 0$ over edge $e = \{v, w\}$ and assume that the dispatch is successful so that κ is acquired by node w in round t. If w does not experience a restart in round t, then the δ-component D_κ is updated so that (1) its sole self loop moves from $(v, v) \in E_\delta^t$ to $(w, w) \in E_\delta^{t+1}$; and (2) the direction of e changes from $(w, v) \in E_\delta^t$ to $(v, w) \in E_\delta^{t+1}$.

The Main Theorems. The analysis of our algorithm stands on three main theorems.

▶ **Theorem 1.** *Each node in V may experience at most one restart throughout the execution. Moreover, there exists some $t_r^* = O(N)$ such that from time t_r^* onward, (1) all nodes are covered and do not experience any restart; and (2) no token dies. This means in particular that if a token κ is alive at time $t \geq t_r^*$ with $\mathcal{V}(D_\kappa^t) = V$, then κ is alive at time t' with $\mathcal{V}(D_\kappa^{t'}) = V$ for every $t' > t$.*

The time t_r^* promised in Thm. 1 can be thought of as the time at which the algorithm is guaranteed to recover from the faults that the initial configuration may exhibit. To establish this theorem, we first prove that a token that completes a *natural traversal* – namely, a traversal that has been invoked as part of the execution, rather that being sprouted from the initial configuration – must be strong. Next, we prove that a token that completed a natural traversal and turned cold, remains strong until the next time it turns hot. Finally, we prove that after $O(N)$ time, any strong token that starts a traversal is guaranteed to complete the traversal (without dying in the middle). Refer to [44] for a full proof.

▶ **Theorem 2.** *Let $t_s^* \geq t_r^*$ be the earliest time such that exactly one token is alive at time t_s^*. Then $t_s^* = O(N \log^2 N)$ in expectation and whp.*

Thm. 2 establishes the stabilization properties of our algorithm: Recall that after time t_r^*, a single token is alive if and only if this token covers the whole graph. Therefore, from time t_s^* onward, the graph admits no crossing edges and any invocation of `Propose` excludes the root transfer and proposing epochs. This means that the `prnt` and `chld` variables of the nodes in V remain unchanged and, in particular, the (single) root remains fixed. The main probabilistic argument behind the proof of Thm. 2 is that if a token κ starts a phase ϕ after time t_r^* and it is not the last remaining token in the graph, then with probability low-bounded by a positive constant, κ is dissolved when ϕ ends. Again, the reader is referred to [44] for a full proof.

▶ **Theorem 3.** *Until it stabilizes, the algorithm sends $O(N \log^2 N)$ messages in expectation and whp. After stabilizing, the algorithm sends at most one message per round.*

Proof. Node $v \in V$ sends at most one message per round, thus $O(N)$ messages are sent in round 0. In rounds $t > 0$, node v may send a message only if it holds a token, thus establishing the desired bound on the message complexity after stabilization, when one token is alive. It remains to bound the number of messages sent during the time interval $[1, t_s^*)$. This is done separately for each message type.

Consider a `pass_tkn` message M sent from node $v \in V$ to a node $u \in N(v)$. If u ignores M, then the token that M carries dies. Thm. 1 ensures that the total number of distinct tokens that existed throughout the execution is up-bounded by $2N$, hence $O(N)$ `pass_tkn` messages are ignored throughout the execution; we focus hereafter on the `pass_tkn` messages that are not ignored.

Consider a node $v \in V$. Recalling that v suffers a premature discovery (and invokes `Restart`) if it receives an (unignored) discovery `pass_tkn` message while $v.\texttt{tmr} < C_{\text{tr}}N$, we conclude that v receives $O(1)$ discovery `pass_tkn` messages throughout any time interval of length $C_{\text{tr}}N$. Moreover, v must receive a discovery `pass_tkn` message between any two retraction `pass_tkn` messages it sends, thus v sends $O(1)$ retraction `pass_tkn` messages throughout any time interval of length $C_{\text{tr}}N$.

Since any `pass_tkn` message sent over edge $e \in E$ is either a discovery message or a retraction message of one of e's endpoints, it follows that we can charge all the (unignored) `pass_tkn` messages sent throughout the execution to the nodes so that each node $v \in V$ is charged with $O(1)$ messages in the time interval $[1 + iC_{\text{tr}}N, 1 + (i+1)C_{\text{tr}}N)$ for any $i \in \mathbb{Z}_{\geq 0}$. Therefore, the total number of `pass_tkn` messages sent during the time interval $[1, t_s^*)$ is $O(N) + O(N) \cdot \frac{t_s^*}{N} = O(N + t_s^*)$.

An upper bound of $O(N + t_s^*)$ on the number of `root_trns` (resp., `propose`) messages sent during $[1, t_s^*)$ follows from the observation that a node must receive (and send) at least one `pass_tkn` message between any two `root_trns` (resp., `propose`) messages it sends. Finally, each `accept` message can be charged to either a `propose` message or to a dying token; the latter accounts to $O(N)$ additional messages.

To sum up, the total number of messages sent during the time interval $[1, t_s^*)$ is $O(N + t_s^*)$. Thm. 3 follows from Thm. 2 ensuring that $t_s^* \leq O(N \log^2 N)$ in expectation and whp. ◀

References

1 Yehuda Afek and Anat Bremler. Self-stabilizing unidirectional network algorithms by power-supply. In *SODA*, volume 97, pages 111–120, 1997.

2 Yehuda Afek and Geoffrey M Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7(1):27–34, 1993.

3 Yehuda Afek and Eli Gafni. Time and message bounds for election in synchronous and asynchronous complete networks. *SIAM Journal on Computing*, 20(2):376–394, 1991.

4 Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient self stabilizing protocols for general networks. In *International Workshop on Distributed Algorithms*, pages 15–28. Springer, 1990.

5 Yehuda Afek, Shay Kutten, and Moti Yung. The local detection paradigm and its applications to self-stabilization. *Theoretical Computer Science*, 186(1-2):199–229, 1997.

6 Marcos K Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Stable leader election. In *International Symposium on Distributed Computing*, pages 108–122. Springer, 2001.

7 Thamer Alsulaiman, Andrew Berns, and Sukumar Ghosh. Low-communication self-stabilizing leader election in large networks. In Teruo Higashino, Yoshiaki Katayama, Toshimitsu Masuzawa, Maria Potop-Butucaru, and Masafumi Yamashita, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 348–350, 2013.

8 Karine Altisen, Stéphane Devismes, Swan Dubois, and Franck Petit. Introduction to distributed self-stabilizing algorithms. *Synthesis Lectures on Distributed Computing Theory*, 8(1):1–165, 2019.

9 Anish Arora and Mohamed Gouda. Distributed reset. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 316–331. Springer, 1990.

10 Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

11 Baruch Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 230–240, 1987.

12 Baruch Awerbuch, Israel Cidon, and Shay Kutten. Communication-optimal maintenance of replicated information. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pages 492–502. IEEE, 1990.

13 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.

14 Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. Time optimal self-stabilizing synchronization. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 652–661, 1993.

15 Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction. In *FOCS*, pages 268–277, 1991.

16 Baruch Awerbuch, Boaz Patt-Shamir, George Varghese, and Shlomi Dolev. Self-stabilization by local checking and global reset. In *International Workshop on Distributed Algorithms*, pages 326–339. Springer, 1994.

17 Baruch Awerbuch and George Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 258–267. IEEE, 1991.

18 Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. Memory space requirements for self-stabilizing leader election protocols. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 199–207, 1999.

19 Lélia Blin, Maria Potop-Butucaru, Stephane Rovedakis, and Sébastien Tixeuil. A new self-stabilizing minimum spanning tree construction with loop-free property. In *International Symposium on Distributed Computing*, pages 407–422. Springer, 2009.

20 Lélia Blin and Sébastien Tixeuil. Compact self-stabilizing leader election for general networks. *Journal of Parallel and Distributed Computing*, 144:278–294, 2020.

21 Janna Burman and Shay Kutten. Time optimal asynchronous self-stabilizing spanning tree. In *International Symposium on Distributed Computing*, pages 92–107. Springer, 2007.

22 Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

23 Miguel Castro, Peter Druschel, A-M Kermarrec, and Antony IT Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications*, 20(8):1489–1499, 2002.

24 Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, 2007.

25 Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

26 Johanne Cohen, Jonas Lefèvre, Khaled Maamra, George Manoussakis, and Laurence Pilard. The first polynomial self-stabilizing 1-maximal matching algorithm for general graphs. *Theoretical Computer Science*, 782:54–78, 2019.

27 Zeev Collin and Shlomi Dolev. Self-stabilizing depth-first search. *Information Processing Letters*, 49(6):297–301, 1994.

28 Alain Cournier, Stéphane Rovedakis, and Vincent Villain. The first fully polynomial stabilizing algorithm for bfs tree construction. *Information and Computation*, 265:26–56, 2019.

29 Ajoy K. Datta, Colette Johnen, Franck Petit, and Vincent Villain. Self-stabilizing depth-first token circulation in arbitrary rooted networks. *Distributed Computing (DC)*, 13(4):207–2018, 2000.

30 Ajoy K. Datta, Lawrence L. Larmore, and Toshimitsu Masuzawa. Communication efficient self-stabilizing algorithms for breadth-first search trees. In *International Conference On Principles Of DIstributed Systems*, pages 293–306. Springer, 2014.

31 Ajoy K Datta, Lawrence L Larmore, and Priyanka Vemula. A self-stabilizing o (k)-time k-clustering algorithm. *The Computer Journal*, 53(3):342–350, 2010.

32 B DeCleene, L Dondeti, S Griffin, T Hardjono, D Kiwior, J Kurose, D Towsley, S Vasudevan, and C Zhang. Secure group communications for wireless networks. In *2001 MILCOM Proceedings Communications for Network-Centric Operations: Creating the Information Force (Cat. No. 01CH37277)*, volume 1, pages 113–117. IEEE, 2001.

33 Carole Delporte-Gallet, Stéphane Devismes, and Hugues Fauconnier. Robust stabilizing leader election. In *SSS'07*, volume 4838, pages 219–233. Springer, 2007. `doi:10.1007/978-3-540-76627-8_18`.

34 Stéphane Devismes and Colette Johnen. Silent self-stabilizing bfs tree algorithms revisited. *Journal of Parallel and Distributed Computing*, 97:11–23, 2016.

35 Stéphane Devismes, Toshimitsu Masuzawa, and Sébastien Tixeuil. Communication efficiency in self-stabilizing silent protocols. In *ICDCS'09*, pages 474–481. IEEE, 2009. `doi:10.1109/ICDCS.2009.24`.

36 Edsger W Dijkstra. Self-stabilization in spite of distributed control. In *Selected writings on computing: a personal perspective*, pages 41–46. Springer, 1982.

37 Danny Dolev, Maria Klawe, and Michael Rodeh. An o (n log n) unidirectional distributed algorithm for extrema finding in a circle. *Journal of algorithms*, 3(3):245–260, 1982.

38 Shlomi Dolev. *Self-stabilization*. MIT Press, Cambridge, MA, USA, 2000.

39 Shlomi Dolev, Mohamed G Gouda, and Marco Schneider. Memory requirements for silent stabilization. *Acta Informatica*, 36(6):447–462, 1999.

40 Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago J. Theor. Comput. Sci.*, 1997, 1997.

41 Shlomi Dolev, Amos Israeli, and Shlomo Moran. Resource bounds for self stabilizing message driven protocols. In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 281–293, 1991.

42 Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7(1):3–16, 1993.

43 Anaïs Durand and Shay Kutten. Reducing the number of messages in self-stabilizing protocols. In *International Symposium on Stabilizing, Safety, and Security of Distributed Systems*, pages 133–148. Springer, 2019.

44 Xavier Défago, Yuval Emek, Shay Kutten, Toshimitsu Masuzawa, and Yasumasa Tamura. Communication efficient self-stabilizing leader election (full version), 2020. `arXiv:2008.04252`.

45 Michael Elkin. A simple deterministic distributed mst algorithm with near-optimal time and message complexities. *Journal of the ACM (JACM)*, 67(2):1–15, 2020.

46 Greg N Frederickson and Nancy A Lynch. Electing a leader in a synchronous ring. *Journal of the ACM (JACM)*, 34(1):98–115, 1987.

47 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.

48 Mohsen Ghaffari and Fabian Kuhn. Distributed mst and broadcast with fewer messages, and faster gossiping. In *32nd International Symposium on Distributed Computing (DISC 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

49 Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

50 Robert Gmyr and Gopal Pandurangan. Time-message trade-offs in distributed algorithms. *arXiv preprint arXiv:1810.03513*, 2018.

51 Kostas P Hatzis, George P Pentaris, Paul G Spirakis, Vasilis T Tampakas, and Richard B Tan. Fundamental control algorithms in mobile networks. In *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, pages 251–260, 1999.

52 Wendi Rabiner Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *Proceedings of the 33rd annual Hawaii international conference on system sciences*, pages 10–pp. IEEE, 2000.

53 Lisa Higham and Zhiying Liang. Self-stabilizing minimum spanning tree construction on message-passing networks. In *International Symposium on Distributed Computing*, pages 194–208. Springer, 2001.

54 Shing-Tsaan Huang and Nian-Shing Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 7(1):61–66, 1993.

55 Michael Isard. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, 2007.

56 Colette Johnen, Gianluigi Alari, Joffroy Beauquier, and Ajoy K Datta. Self-stabilizing depth-first token passing on rooted networks. In *International Workshop on Distributed Algorithms*, pages 260–274. Springer, 1997.

57 Colette Johnen and Joffroy Beauquier. Distributed self-stabilizing depth-first token circulation with constant memory. In *2nd Workshop on Self-Stabilizing System (WSS)*, pages 4–1, 1995.

58 Shmuel Katz and Kenneth J Perry. Self-stabilizing extensions for meassage-passing systems. *Distributed Computing*, 7(1):17–26, 1993.

59 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*, pages 71–80, 2015.

60 Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Towards secure and scalable computation in peer-to-peer networks. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 87–98. IEEE, 2006.

61 Ephraim Korach, Shlomo Moran, and Shmuel Zaks. Tight lower and upper bounds for some distributed algorithms for a complete network of processors. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 199–207, 1984.

62 Alex Kravchik and Shay Kutten. Time optimal synchronous self stabilizing spanning tree. In *International Symposium on Distributed Computing*, pages 91–105. Springer, 2013.

63 Shay Kutten and Dmitry Zinenko. Low communication self-stabilization through randomization. In *International Symposium on Distributed Computing*, pages 465–479. Springer, 2010.

64 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

65 Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

66 Butler W Lampson. How to build a highly available system using consensus. In *International Workshop on Distributed Algorithms*, pages 1–17. Springer, 1996.
67 Gérard Le Lann. Distributed systems - towards a formal approach. In *IFIP Congress*, pages 155–160, 1977.
68 Mikel Larrea, Antonio Fernández, and Sergio Arévalo. Optimal implementation of the weakest failure detector for solving consensus (brief announcement). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, page 334, 2000.
69 Edward K Lee and Chandramohan A Thekkath. Petal: Distributed virtual disks. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 84–92, 1996.
70 Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.
71 Navneet Malpani, Jennifer L Welch, and Nitin Vaidya. Leader election algorithms for mobile ad hoc networks. In *Proceedings of the 4th international workshop on Discrete algorithms and methods for mobile computing and communications*, pages 96–103, 2000.
72 Ali Mashreghi and Valerie King. Broadcast and minimum spanning tree with $o(m)$ messages in the asynchronous congest model. *arXiv preprint arXiv:1806.04328*, 2018.
73 Ali Mashreghi and Valerie King. Brief announcement: Faster asynchronous mst and low diameter tree construction with sublinear communication. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
74 Toshimitsu Masuzawa. Silence is golden: self-stabilizing protocols communication-efficient after convergence. In *Symposium on Self-Stabilizing Systems*, pages 1–3. Springer, 2011.
75 Toshimitsu Masuzawa, Taisuke Izumi, Yoshiaki Katayama, and Koichi Wada. Brief announcement: Communication-efficient self-stabilizing protocols for spanning-tree construction. In *OPODIS'09*, pages 219–224, 2009.
76 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 SIGACT Symposium on Theory of Computing*, pages 743–756, 2017.
77 Boaz Patt-Shamir and Mor Perry. Proof-labeling schemes: Broadcast, unicast and in between. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 1–17. Springer, 2017.
78 Charles E Perkins and Elizabeth M Royer. Ad-hoc on-demand distance vector routing. In *Proceedings WMCSA'99. Second IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100. IEEE, 1999.
79 Radia Perlman. *Interconnections: bridges, routers, switches, and internetworking protocols*. Addison-Wesley Professional, 2000.
80 Franck Petit. Fast self-stabilizing depth-first token circulation. In *International Workshop on Self-Stabilizing Systems*, pages 200–215. Springer, 2001.
81 Sudarshan Vasudevan, Jim Kurose, and Don Towsley. Design and analysis of a leader election algorithm for mobile ad hoc networks. In *Proceedings of the 12th IEEE International Conference on Network Protocols, 2004. ICNP 2004.*, pages 350–360. IEEE, 2004.