FireLedger: A High Throughput Blockchain Consensus Protocol

Yehonatan Buchnik
CS Technion
yon_b@cs.technion.ac.il

Roy Friedman
CS Technion
roy@cs.technion.ac.il

ABSTRACT

Blockchains are distributed secure ledgers to which transactions are issued continuously and each block of transactions is tightly coupled to its predecessors. Permissioned blockchains place special emphasis on transactions throughput. In this paper we present FireLedger, which leverages the iterative nature of blockchains in order to improve their throughput in optimistic execution scenarios. FireLedger trades latency for throughput in the sense that in FireLedger the last $f + 1$ blocks of each node's blockchain are considered tentative, i.e., they may be rescinded in case one of the last $f + 1$ blocks proposers was Byzantine. Yet, when optimistic assumptions are met, a new block is decided in each communication step, which consists of a proposer that sends only its proposal and all other participants are sending a single bit each. In our performance study FireLedger obtained $20\% - 600\%$ better throughput than state of the art protocols like HotStuff and BFT-SMaRt, depending on the configuration.

PVLDB Reference Format:
Yehonatan Buchnik and Roy Friedman. FireLedger: A High Throughput Blockchain Consensus Protocol. *PVLDB*, 13(9): 1525-1539, 2020.
DOI: https://doi.org/10.14778/3397230.3397246

1. INTRODUCTION

Blockchains are becoming popular in many areas such as cryptocurrencies, supply-chains, insurance, and others [12]. Blockchains are often characterized as either *unpermissioned* or *permissioned*. In permissioned mode, the blockchain is executed among a set of n known participants under the assumption that at most f of them are faulty [29]. In this setting, blockchain becomes a special case of traditional *replication state machine* (RSM) [58, 70]. A common approach to implementing RSM is by repeatedly running a consensus protocol to decide on the next transaction to be executed [59] with the optimization of batching multiple transactions in each invocation of the consensus protocol [49].

Proceedings of the VLDB Endowment, Vol. 13, No. 9
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3397230.3397246

The assumed possible type of failures affects the type of consensus protocols that are used. *Benign* failures such as a node crash and occasional message omission can be overcome by benign consensus protocols, e.g., [32, 53, 59, 65]. On the other hand, *Byzantine* failures [60] in which a faulty node may arbitrarily deviate from its code require Byzantine fault tolerant protocols (BFT), e.g., [18, 30, 56]. In the totally asynchronous case, the seminal FLP result showed that even benign consensus cannot be solved [46]. Yet, when enriching the environment with some minimal eventual synchrony assumptions, e.g., partial synchrony [39, 42], or with unreliable failure detector oracles [33], benign consensus becomes solvable when $f < n/2$ and Byzantine consensus requires $f < n/3$. This is as long as the network does not become partitioned [21].

We focus on permissioned blockchains assuming Byzantine failures and partial synchrony. According to a recent survey by PwC [12] only a third of companies currently using or planing to use blockchains intend on using unpermissioned blockchains. Many recent unpermissioned proposals, e.g., Algorand [51], Tendermint [25], Thunderella [66], and HoneyBudger [63], can be viewed as running a permissioned protocol coupled with a higher level meta-protocol that continuously selects which nodes can participate in the internal permissioned one. Variants of this approach are sometimes referred to as *delegated proof of stake (dPoS)* and *Proof of Authority* (PoA). Hence, any improvement in permissioned protocols will likely yield better unpermissioned protocols.

Many such works try to optimize performance in the "common case" in which there are no failures and the network behaves in a synchronous manner. These are likely to be common in permissioned blockchains, e.g., executed between major financial institutions, established business partners, etc. Yet, all the above mentioned works run each protocol instance for completion. Alas, we claim that in a production blockchain system, where transactions are being submitted continuously for as long as the service exists, there is potential for reducing the per transaction and per block communication overhead. This is by assuming optimistically that the initial proposer of each consensus invocation is correct, and only performing a recovery phase periodically for a batch of affected consensus invocations and only if it is needed.

Our Contributions. We propose FireLedger, a new communication frugal optimistic permissioned blockchain protocol. FireLedger utilizes the rotating proposer scheme while optimistically assuming that the proposer is correct and that the environment behaves synchronously. This is supported

by our novel *Weak Reliable Broadcast* (WRB) abstraction. With WRB, nodes only agree on whether to (tentatively) accept a block proposed by the proposer, without agreeing on the content of the block (in case the proposer is Byzantine). Specifically, if these assumptions are violated, we do not insist on enforcing agreement immediately. Instead, we rely on the fact that at least one out of every $f+1$ proposers is correct. When a correct node discovers, using blockchain's authentication data, that any of the last $f+1$ blocks was not decided correctly in the initial transmission phase, it runs a combined recovery phase for all these incorrectly executed invocations. This is by invoking a full Byzantine consensus protocol. At the end of this combined recovery phase, it is ensured that the current prefix of the blockchain is agreed by all correct nodes and will never change as long as there are at most f Byzantine failures. A single recovery phase may decide the last f blocks, thereby amortizing its cost.

The main benefit of our approach is that when the optimistic assumption holds, the communication overhead of deciding on a block involves a single proposer broadcasting its block and all other nodes broadcasting a single bit of unsigned protocol data (WRB). Further, a new block is being decided in each communication step. This is by leveraging the iterative nature of blockchain as well as the authentication data that is associated with each block header.

Notice that f is an upper bound on the maximal number of Byzantine nodes in the system. Yet, in many permissioned blockchains settings, nodes are likely to be highly secured. Further, in our protocol any Byzantine deviation from the protocol results in a strong proof of which node was the culprit. Hence, we expect that in real deployments the optimistic assumptions will hold almost always. In particular, once a proof of Byzantine behavior is being generated, the corresponding Byzantine node will be removed from the system, often resulting in financial penalties and loss of face for the owner of this node.

The price paid by our algorithm is that finality of a decision is postponed for $f+1$ invocations (or blocks). That is, we trade bandwidth and throughput for latency of termination. As we show when evaluating the performance of our protocol, the average termination latency of blocks is at most a few seconds. In return, when running on non-dedicated virtual machines and network, in a single Amazon data-center, we demonstrate performance of up to 160K transactions per seconds (tps). In a non-dedicated multi data-center settings, we obtain up to 30K tps.

2. RELATED WORK

Optimistic Consensus: Two main methods were suggested for designing optimistic consensus protocols: (i) satisfying safety from the nodes' point of view [48, 57, 62] or (ii) satisfying safety only from the clients' point of view [13, 18, 56]. In the first approach, to detect inconsistencies, nodes must continuously update other nodes with their state (the exception is [48] that uses randomization). In the second approach, nodes are allowed to be temporarily inconsistent. Only when a client detects an inconsistency, e.g., by receiving inconsistent replies, it initiates a special recovery mechanism to restore the system's consistency. Also, in protocols like [56], a single slow replica causes the protocol to switch to its slow path mode. Concerning blockchains, the first method ignores blockchain's unique features that can be leveraged. In contrast, running the blockchain nodes as clients of an agreement service results in at least two communication steps protocol even in the "good cases".

Blockchain Systems: Most permissioned blockchain protocols assume a partially synchronous network while utilizing traditional BFT concepts. Such platforms run a more computationally efficient protocol than unpermissioned blockchains but require an a-priori PKI infrastructure. Traditional BFT solutions are not scalable in the number of participants [74] as their communication complexity grows quadratically in the number of nodes. Hence, such solutions focus on (i) sharding the execution's roles between multiple layers, leaving the consensus to be run by a small set of nodes, and on (ii) designing optimized dedicated BFT consensus protocols. Known platforms like HLF [3, 5, 16, 72] and R3 Corda [2] offer new models of layered computation and run the BFT-SMaRt [20] protocol, or a variant of it, for ordering.

Platforms such as Chain Core [1, 31], Iroha [6], Symbiont-Assembly [8] and Tendermint [9] offer new optimized BFT Consensus algorithms. Iroha, inspired by the original HLF (v0.6) architecture, runs the Sumeragi consensus protocol which is heavily inspired by BChain [41]. BChain is a chain-replication system in which n nodes are linearly arranged and a transaction is moved among the nodes in a chain topology. Namely, each node normally receives a message only from its predecessor. Like FireLedger, BChain trades latency for throughput and it has the potential to achieve the best possible throughout [18, 52]. Unlike FireLedger, BChain's latency is bounded by at least n rounds. Symbiont-Assembly implements its own variant of BFT-SMaRt. Tendermint implements an iterative variant of PBFT [30] designed by Buchman et el. [25]. Chain Core runs the Federated consensus protocol in which one node is the leader and n are validators. This protocol is Byzantine resilient for $f < \frac{n}{3}$ only as long the leader is correct. Red Belly blockchain [7] offers both, a new computation model that balances the verification load among *verifies* nodes and the Democratic BFT consensus [37] that is able to scale the throughput with the number of proposers. Finally, Honey-Badger BFT (HBB) [63, 4] is a randomized protocol .

HotStuff [75] extends transactions' finality to 3 rounds and employs signature aggregation [22] in order to obtain linear communication overhead. HotStuff requires all nodes to sign an asymmetric signature on each block in the optimistic case while in FireLedger this is done only by the proposer that generated the block. Since signing takes pure CPU time, fewer asymmetric signatures enable better throughput.

The goal of StreamChain [54] is to dramatically reduce transactions finality times in permissioned blockchains. This is done by adopting a streaming architecture and utilizing batching (blocks) only to amortize disk writing times of committed transactions. The finality times obtained with StreamChain are below 10 ms and throughput of $1,500$ tps.

BlockchainDB [44] builds a database layer on top of a sharded blockchain. The latter is used as a trusted decentralized data-store. This design enjoys improved performance for applications that share data on blockchains and benefits from a standard DB query language capabilities.

3. PRELIMINARIES

3.1 System Model

We consider an asynchronous fully connected environment consisting of n nodes out of which at most $f < \frac{n}{3}$ may incur

Byzantine failures [17, 67]. Asynchronous means that no upper bound on the messages' transfer delays exists and nodes have no access to a global clock. Fully connected means that any two nodes are connected via a reliable link. Reliable means that a link does not lose, modify or duplicate a sent message. Notice that unreliable fair lossy links can be transformed into reliable ones using sequence numbering, retransmissions, and error detection codes [69]. A Byzantine failure means that a node might deviate arbitrarily w.r.t. its protocol code including, e.g., sending arbitrary messages, sending messages with different values to different nodes, or failing to send any or all messages. Yet, we assume that nodes cannot impersonate each other. A node suffering from a Byzantine failure at any point during its operation is called *Byzantine*; otherwise, it is said to be *correct*.

In order to circumvent the FLP result [46], we enrich the system with the $\diamond Synch$ assumption [23]. That is, after an unknown time τ there is an unknown upper bound δ on a message's transfer delay. As in most Byzantine fault tolerance works [18, 30, 56], $\diamond Synch$ is only needed to ensure liveness, meaning that even under severe network delays safety is never violated[1]. Finally, a node may sign a message by an unforgeable signature. We denote the signature of node p on message m by $sig_p(m)$. The implementation of the signature mechanism is done by a well known cryptographic technique, such as symmetric ciphers [43], RSA [68] or an elliptic curves digital signature (ECDS) [55].

3.2 Underlying Protocols

The following serve as building blocks for FireLedger.

Reliable Broadcast (RB). The reliable broadcast abstraction [24] (denoted **RB-Broadcast**) ensures reliable message delivery in the presence of Byzantine failures. To utilize **RB-Broadcast** nodes may invoke two methods: *RB-broadcast* and *RB-deliver*. A correct node that wishes to broadcast a message m invokes *RB-broadcast(m)* while a node that expects to receive a message invokes *RB-deliver*. By a slight abuse of notation, we denote *RB-deliver(m)* the fact than an invocation of *RB-deliver* returned the message m and say that the invoking process has RB-delivered m. The **RB-Broadcast** abstraction satisfies the following:

RB-Validity: If a correct node has RB-delivered a message m from a correct node p, then p has invoked *RB-broadcast(m)*.

RB-Agreement: If a correct node *RB-delivers* a message m, then all correct nodes eventually *RB-deliver* m.

RB-Termination: If a correct node invokes *RB-broadcast(m)*, then all correct nodes eventually *RB-deliver* m.

Atomic Broadcast (AB). Atomic broadcast [38] requires in addition to the *RB-Broadcast* properties the **Atomic-Order** property, i.e., all messages delivered by correct nodes are delivered in the same order by all correct nodes.

[1]With the $\diamond Synch$ property, Byzantine nodes can continue reordering messages forever, however after a bounded unknown time (GST), all messages sent by correct nodes arrive within a bounded latency despite Byzantine activity.

(Optimistic) Binary Byzantine Consensus. *Binary Byzantine Consensus* (BBC) is the simplest variant of *Multi-value Byzantine Consensus* (MVC) [60] in which only two values are possible. A solution to the BBC problem satisfies the following properties [42, 36]:

BBC-Validity: If all correct nodes have proposed the same value v, then v must be decided.

BBC-Agreement: No two correct nodes decide differently.

BBC-Termination: Each correct node eventually decides.

As there are only two possible values in BBC, an *Optimistic BBC* (OBBC) is capable of achieving an agreement in a single communication step if a predefined set of favorable conditions are met [26, 27, 48, 64].

3.3 Problem Statement

Blockchain algorithms require an external validity mechanism as sometimes even Byzantine nodes may propose legal values (or blocks) [61]. Therefore, the validity of a value may be defined by an external predefined method. The *Validity Predicate-based Byzantine Consensus* (VPBC) [28, 37] abstraction captures this observation by replacing the validity property with **VPBC-Validity**, i.e., a decided value satisfies an external predefined VALID method.

Recall that our goal is to enable a relaxed form of finality in exchange for better throughput and fast light weight agreement when the optimistic assumptions hold. To that end, we present the following definitions: In a blockchain, each block carries a glimpse to its creator knowledge of the system's state. This glimpse is encapsulated in the hash that each block carries. In order to leverage the iterative nature of blockchains, we define a weaker model than *VPBC* in which we denote each iteration with a round number r. Next, we define the following per round notions:

Tentative decision: A decision of the protocol at a given node and round that might still be changed.

Definite decision: A decision of the protocol at a given node and round that will never change.

v_p^r: A value that was decided or received by p in round r of the protocol. v^r denotes a value that was decided or received by some node in round r.

$d(v_p^r)$: Let r' be the current round of the protocol that node p runs. For a given v_p^r (possible tentative), we denote its *depth* as $d(v_p^r) = r' - r$.

DEFINITION 3.3.1. *Blockchain Based Finality Consensus (BBFC) Let* VALID *be a predefined method as in VPBC and let ρ be a predefined fixed constant. The* ρ-Blockchain Based Finality Consensus *(BBFC(ρ)) abstraction defines the following properties:*

BBFC-Validity: *A decided (possible tentative) value v satisfies the* VALID *method.*

BBFC-Agreement: *For any two correct nodes p, q with v_p^r, v_q^r as their decided value in round r. If both $d(v_p^r) > \rho$ and $d(v_q^r) > \rho$ then $v_p^r = v_q^r$.*

BBFC-Termination: *Every round eventually terminates.*

BBFC-Finality: *In every round $r' > r + \rho$, v_p^r is definite.*

BBFC guarantees the VPBC's properties only for decisions at depth greater than ρ. With blockchains, as every block contains an authentication data regarding its predecessors, it provides a cryptographic summary of its creator history. This information assists in detecting failures without the necessity of sending more information. In addition, blocks are continuously added to the chain. Thus, a block eventually becomes deep enough such that it satisfies the standard VPBC properties. Let us note that the BBFC-Agreement property is similar to the notion of *common prefix* in [50].

In a blockchain setting, *clients* of the system submit *transactions* to the nodes, and the decisions values are blocks, each consisting of zero or more transactions previously submitted by clients. In case the VALID method may accept empty blocks, we would like to prevent trivial implementations in which every node locally generates empty blocks continuously. Obviously, the throughput of such a protocol would be 0, and thus it would be considered useless. Yet, in order to prove that a protocol does not unintentionally suffer from such a behavior even under Byzantine failures we add the following requirement:

Non-Triviality: If clients repeatedly submit transactions to the system, then the nodes repeatedly decide definitively non-empty blocks.

4. WEAK RELIABLE BROADCAST

4.1 Overview

The *Weak Reliable Broadcast* (WRB) abstraction serves as FireLedger's main message dissemination mechanism. It offers weaker agreement guarantees than Bracha's **RB-Broadcast** [24]. In general, WRB ensures that the nodes agree on (*i*) the sender's identity and (*ii*) whether to deliver a message at all, rather than the content of the message[2]. WRB is associated with the *WRB-broadcast* and *WRB-deliver* methods. A node that wishes to disseminate a message m invokes *WRB-broadcast(m)*. If a node expects to receive a message m from k through this mechanism it invokes *WRB-deliver(k)*. *WRB-deliver(k)* returns a message m where m is the received message. If the nodes were not able to deliver k's message, *WRB-deliver(k)* returns *nil*. Formally, WRB satisfies the following properties:

WRB-Validity: If a correct node *WRB-delivered(k)* $m \neq nil$, then k has invoked *WRB-broadcast(m)*.

WRB-Agreement: When two correct nodes p, q *WRB-delivered(k)* m_p, m_q respectively from k, then either $m_p = m_q = nil$ or $m_p \neq nil \wedge m_q \neq nil$.

WRB-Termination: If a correct node p *WRB-deliver(k)* m from k, then every correct node that is trying to *WRB-deliver(k)* eventually *WRB-deliver(k)* some message m' from k.

WRB-Non-Triviality: If a correct node k repeatedly invokes *WRB-broadcast(m)* then eventually all correct nodes will *WRB-deliver(k)* m.

[2]To the best of our knowledge, we are the first to discuss WRB. Bracha's approach [24] can be viewed as the opposite, first agree on the content of the message (*consistent broadcast*), and then agree whether it should be delivered.

```
[1]  timer ← τ;
[2]  Procedure WRB-broadcast(m)
[3]  |  broadcast (m, sig_p(m)); /* push phase     */
[4]  Procedure WRB-deliver(k)
[5]  |  start timer;
[6]  |  /* timer's value is set in lines 1, 14
     |     and 19 of WRB-deliver's last
     |     invocation                              */
[7]  |  wait until a valid (m, sig_k(m)) has been
     |   received or timer has expired ;
[8]  |  if a valid (m, sig_k(m)) has been received then
[9]  |  |  d ← OBBC.propose(1);
[10] |  |  /* If no node has proposed 0, then
     |  |     OBBC.propose ends in a single
     |  |     communication step                   */
[11] |  else
[12] |  |  d ← OBBC.propose(0);
[13] |  end
[14] |  increase timer;
[15] |  if d = 0 then
[16] |  |  return nil;
[17] |  end
[18] |  if a valid (m, sig_k(m)) has been received then
[19] |  |  adjust timer;
[20] |  |  return m;
[21] |  end
[22] |  broadcast(REQ, k);
[23] |  wait until a valid (m', sig_k(m')) has been
     |   received;
[24] |  return m';
[25] |  upon receiving (REQ, k) from q∧ a valid
     |   (m, sig_k(m)) has been received do
[26] |  |  send(m, sig_k(m))) to q;
[27] |  end
```

Algorithm 1: Weak Reliable Broadcast - code for p

By FireLedger's use of WRB-broadcast, outlined in Section 5, it is possible that messages of a given leader are delayed to the point they are not delivered, i.e., the WRB-broadcast protocol ends by delivering an empty message. However, due to the WRB-broadcast protocol, either all correct nodes deliver a message that was sent by the leader (perhaps not the same one at all nodes), or none does so. The correctness of the below implementation of WRB-broadcast is independent of any timing assumptions. The performance benefits of WRB for FireLedger are discussed in Section 5.4.

4.2 Implementing WRB

The pseudo-code implementation of WRB is listed in Algorithm 1. To *WRB-broadcast* a message, a node simply broadcasts it to everyone (line 3). When p invokes *WRB-deliver(k)* it performs the following:

▷ It waits for at most *timer* to receive k's message (line 7).

▷ If such a valid message has been received, then p votes to deliver it using an OBBC protocol. Else, p votes against delivering k's message (lines 8-13). Recall that if no node has proposed 0, OBBC ends in a single communication step.

▷ If the decision is not to deliver (OBBC returned 0), p returns *nil* and increases the timer (lines 14-17).

▷ If it is decided to deliver the message (OBBC returned 1) and the message has already been received by p, then p adjusts the timer and returns m (lines 18-21).

▷ Else, OBBC decided 1, meaning that at least one cor-

rect node received k's message and voted for its acceptance. Thus, p moves to a *pull* phase and pulls k's message from the nodes who did receive it. When p eventually receives a valid message, p returns it (lines 22- 24).
▷ Upon receiving q's request for k's message, if p has k's message m, it sends $(m, sig_k(m))$ to q (lines 25-27).

To ensure liveness, the *timer* is increased each time p does not receive the message (line 14). To avoid having too long timers for too long, *timer* is adjusted downward when a message is received by p (line 19). The exact details are beyond the scope of this paper, but see for example [30].

In a typical implementation of OBBC each node broadcasts its vote [27, 67]. Then if a node receives enough votes for the same value v to safely decide v after this single communication step, it decides v and returns. We present our own OBBC protocol in the full version of this paper [26].

Since the correctness proof of Algorithm 1 is technical, it is deferred to the full version of this paper [26].

5. THE FireLedger PROTOCOL

We present FireLedger in a didactic way: We first show a two-phased *crash fault tolerant* (CFT) ordering protocol based on WRB. We then improve it to a single-phased protocol. Finally, we extend the protocol to tolerate Byzantine failures. The pseudo-code of the protocol appears in Algorithm 2 and 3. Regularly numbered lines correspond to the CFT aspects of the protocol, while lines prefixed with 'b' are the additions to handle Byzantine failures.

5.1 Simplified CFT FireLedger

As mentioned in Section 4, WRB supports an all or nothing delivery that is blind to the message's content. When there are no Byzantine failures, a node never sends different messages to different nodes. Hence, a simple two phase blockchain protocol would be a round based design in which a deterministically selected leader disseminates its block proposal to all nodes using WRB. In case all nodes deliver the proposed block, then this becomes the next block.

Given the continuous iterative nature of blockchains, we may improve the algorithm's round latency to an amortized single phase. For this, we piggyback the $r + 1^{th}$ block on top of the first message that *WRB-deliver* sends when trying to deliver the r^{th} block. We support this by augmenting the *WRB-deliver* method to receive two parameters, *WRB-deliver(k,pgd)*, such that *pgd* is the potential piggybacked data (can be *nil*). The OBBC protocol is also augmented to receive *pgd* and piggyback it on the first message it broadcasts. Recall that we assume an OBBC protocol that always starts by having each node broadcast its vote. Hence in the augmented protocol, each node that starts the OBBC protocol broadcasts its vote alongside the piggybacked *pgd* message, which is made available to the calling code together with the decision value.

The details appear in Algorithm 2. Specifically, on each round r the algorithm performs the following:
▷ If p_i is $(r + 1)$'s proposer, it prepares a new block (lines 12-14). To ensure liveness, if in the previous iteration WRB has failed to deliver a message, then r's proposer also prepares a block and WRB-broadcast it (lines 6-11).
▷ Meanwhile, all nodes are trying to WRB-deliver r's block (line 15). Note that the $r + 1^{th}$ proposer piggybacks the next block on top of this message.

```
 1  r_i ← 0;
 2  proposer_{r_i} ← p_0;
 3  full_mode ← true;
 4  while true do
 5  |  b ← nil;
[b1]|  while proposer_{r_i}'s block was tentatively
    |   decided in the last f rounds do
[b2]|  |  proposer_{r_i} ← (proposer_{r_i} + 1) mod n;
[b3]|  end
 6  |  if i = proposer_{r_i} ∧ full_mode = true then
 7  |  |  /* executed if nil has been
    |  |     WRB-delivered in the last
    |  |     iteration                    */
 8  |  |  b ← prepared block;
 9  |  |  WRB-broadcast(b);
10  |  |  b ← nil;
11  |  end
12  |  if (proposer_{r_i} + 1) mod n = i then
13  |  |  b ← prepared block;
14  |  end
15  |  b_i^{r_i} ← WRB-deliver(proposer_{r_i}, b);
16  |  if b_i^{r_i} = nil then
17  |  |  full_mode ← true;
18  |  |  proposer_{r_i} ← (proposer_{r_i} + 1) mod n;
19  |  |  continue;
20  |  end
21  |  full_mode ← false;
[b4]|  if b_i^{r_i} is not valid then
[b5]|  |  /* validating the block hash against
    |  |     the previous block           */
[b6]|  |  proof ← (b_i^{r_i}, sig_{proposer_{r_i}}(b_i^{r_i}),
    |  |    b_i^{r_i-1}, sig_{proposer_{r_i-1}}(b_i^{r_i-1}));
[b7]|  |  RB-broadcast(proof);
[b8]|  |  invoke RECOVERY(r_i, proof);
[b9]|  |  continue;
[b10]|  end
22  |  append b_i^{r_i} to the chain;
[b11]|  decide b_i^{r_i-(f+2)};
23  |  proposer_{r_i} ← (proposer_{r_i} + 1) mod n;
24  |  r_i ← r_i + 1;
25  end
[b12] upon RB-deliver a valid proof ←
      (b^r, sig_{proposer_r}(b^r), b^{r-1}, sig_{proposer_{r-1}}(b^{r-1})) do
[b13]|  invoke RECOVERY(r, proof);
[b14] end
```

Algorithm 2: FireLedger – code for p_i; the lines that start with 'b' depict the BFT additions

▷ If a block $b^r \neq nil$ has been delivered, then b^r is appended to the chain (line 22) and the protocol continues to round $r + 1$ (line 23- 25).
▷ Else, all correct nodes switch proposer and continue to the next try (lines 16- 20).

We prove the correctness of the full BFT protocol in Section 5.3. Note that due to WRB and the piggybacking method, Algorithm 2 establishes a single-phased protocol as long as there are no Byzantine failures, and the failure pattern matches the specific OBBC optimistic pattern.

Figure 1 presents normal case operation. Each optimistic period starts with the current proposer broadcasting its block. Then, on every round, each node broadcasts a single message (as the first OBBC message), except the next proposer that piggybacks the next block on top of that message.

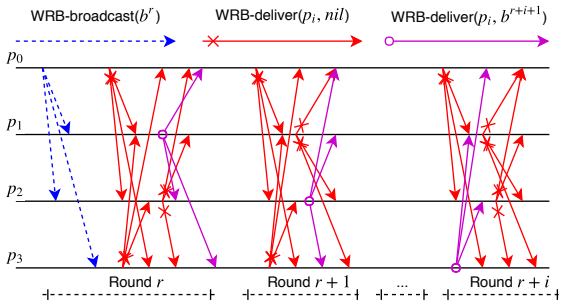

Figure 1: FireLedger in the normal case operation.

```
[1]  Procedure RECOVERY(r, proof)
[2]      versions_r ← {};
[3]      if r_i < r − 1 then
[4]          v ← empty_version;
[5]      else
[6]          v ←
               [(b^{r−(f+1)}, sig_{proposer_{r−(f+1)}}(b^{r−(f+1)})), ...,
               (b^{r−1}, sig_{proposer_{r−1}}(b^{r−1})), ...,
               (b^{r_i}, sig_{proposer_{r_i}}(b^{r_i}))];
[7]      end
[8]      Atomic-broadcast(v);
[9]      repeat
[10]         Atomic-deliver v_j from p_j;
[11]         if v_j is valid then
[12]             versions_r ← versions_r ∪ {v_j};
[13]         end
[14]     until |versions_r| = n − f;
[15]     v' ← the first received among
           {v_j ∈ versions_r | r_j = max{r_k | v_k ∈
           versions_r ∧ (b^{r_k}, sig_{proposer_{r_k}}(b^{r_k}) ∈ v_k}};
[16]     adopt v' and update r_i, and proposer_{r_i};
[17]     full_mode ← true;
```

Algorithm 3: Recovery Procedure – code for p_i

5.2 Full BFT FireLedger

To extend the basic FireLedger to handle Byzantine failures, FireLedger utilizes the fact that there is at least one correct node p_c in all sets of $f+1$ different proposers. Since p_c is correct, when p_c's block is WRB-delivered, all correct nodes receive the very same block, including the hash to its predecessor block. When a correct node detects a chain inconsistency (due to the hash that each block carries), it initiates a traditional BFT based recovery procedure. At the end of the recovery phase, all correct nodes synchronize their chain to the same single valid version. For the recovery procedure, FireLedger maintains the following invariant:

INVARIANT 1. *A node p proceeds to the next round of the algorithm only if it knows that at least $f+1$ correct nodes will eventually proceed as well.*

As a consequence of Invariant 1, if a block b^r is at depth $f+1$ in p's local chain, then there are at least $f+1$ correct nodes for which b^r is at depth of at least $f+1$ in their local chains. In principle, preserving Invariant 1 requires waiting to verify that at least f other correct nodes are moving to the next round. Yet, as the communication pattern we described above already includes an all-to-all message exchange[3] in each round (while executing OBBC), it serves as an implicit acknowledgement, so we have a single-phased algorithm when the optimistic assumptions hold.

Recall that line numbers prefixed by 'b' in Algorithm 2 describe the additional actions to accommodate Byzantine failures. Specifically:
▷ First p_i finds, by a pre-defined order, a prosper that has not successfully proposed a block in the last $f+1$ rounds (lines b1- b3).If p_i detects an inconsistency in the chain, it starts the recovery using reliable broadcast (lines b4- b10).
▷ Upon receiving a valid announcement of inconsistency, p_i initiates the recovery procedure (lines b12- b14). Note that the announcement validation is done against digital signatures and the blocks' hashes.

The recovery installs agreement among all correct nodes regarding the longest possible blockchain prefix as detailed in Algorithm 3. Executing RECOVERY by p_i involves:
▷ p_i proposes, using *Atomic-broadcast*, a valid version of the f blocks that are in disagreement (excluding b^r itself) followed by all the newer blocks it knows about (lines 2–8).
▷ Then p_i collects $n-f$ valid versions (including empty ones) and adopts the first longest agreed prefix of the blockchain (lines 9–17).

Finally, as the recovery procedure may alter only the last $f+1$ blocks, the node decides on the block which is in depth of $f+2$ (line b11)

5.3 Correctness Proof

LEMMA 5.3.1. *BBFC($f+1$)-Validty: A decided (possible tentative) value v satisfies the* VALID *method.*

PROOF. From Algorithm 2's code, a value is appended to the blockchain only if it satisfies the VALID method. □

LEMMA 5.3.2. *Every $f+1$ consecutive decided blocks were proposed by $f+1$ different nodes.*

PROOF. By Algorithm 2's code (lines b1–b3), if the current proposer has successfully proposed a block in the last $f+1$ rounds, then its role is switched to a new proposer. □

LEMMA 5.3.3. *At the recovery procedure's end, all correct nodes adopt the same version.*

PROOF. By the **Atomic-Order** property of the atomic broadcast primitive all correct nodes receive the same versions in the same order. Hence, applying the deterministic rule described in Algorithm 2 results in an agreement among the correct nodes. □

LEMMA 5.3.4. *For any two correct nodes p, q with decided values b_p^r, b_q^r in round r, if $d(b_p^r) > f \wedge d(b_q^r) > f$ then $b_p^r = b_q^r$.*

For lack of space, the proof of this lemma is deferred to the full version of this paper [26].

LEMMA 5.3.5. *BBFC($f+1$)-Agreement: For any two correct nodes p, q, let b_p^r, b_q^r be their decided value in round r. If $d(b_p^r) > f+1 \wedge d(b_q^r) > f+1$ then $b_p^r = b_q^r$.*

PROOF. Follows directly from Lemma 5.3.4. □

[3]An "all-to-all exchange" only means that each correct process must send a message to all other processes, yet none of the processes is required to wait for all such messages.

DEFINITION 5.3.1. *Let $b_p^r, b_p^{r'}$ be two decided (possibly tentative) blocks of p such that $r' > r$. $b_p^{r'}$ is* valid *with respect to b_p^r if the sub-chain $[b_p^r, b_p^{r+1}, ..., b_p^{r'}]$ satisfies the predefined* VALID *method. A sub-chain $[b_p^r, b_p^{r+1}, ..., b_p^{r'}]$ is* valid *with respect to b_p^{r-k}, for some $k \leq r$, if sub-chain $[b_p^{r-k}, ..., b_p^r, ..., b_p^{r'}]$ satisfies the predefined* VALID *method and each $f+1$ consecutive blocks are proposed by $f+1$ different proposers.*

LEMMA 5.3.6. *If during the recovery procedure for round r, a correct node p receives a version v from correct node q, then v is valid with respect to $b_p^{r-(f+2)}$.*

PROOF. If the received version is an empty one, it is trivially valid with respect to $b_p^{r-(f+2)}$. Else, by Lemma 5.3.4 all correct nodes agree on $b^{r-(f+2)}$. As q is correct, by Lemma 5.3.1 q appends only valid blocks to its blockchain. Hence, q's version (that starts with $b_q^{r-(f+1)}$) is valid with respect to $b_q^{r-(f+2)}$ which is identical to $b_p^{r-(f+2)}$. □

DEFINITION 5.3.2. *Let r_g be the most advanced round of the algorithm that any correct node runs. We define the group of nodes whose current round is $\in \{r_g, r_g - 1\}$ by front $= \{p | r_p \in \{r_g - 1, r_g\}\}$. When Invariant 1 is kept, there are at least $f+1$ correct nodes in* front.

LEMMA 5.3.7. *While Invariant 1 is kept, a correct node executing the recovery procedure receives $n-f$ valid versions and at least one of them was received from a node in* front.

PROOF. By **RB-Termination** if a correct node p detects an invalid block and invokes the recovery procedure, eventually every correct node will receive p's proof and will invoke the recovery procedure. Following the system model, the ratio between n and f and Lemma 5.3.6, a correct node does not get blocked while waiting for $n - f$ valid versions (part of whom may be empty). By Invariant 1, p receives at least one version from a correct node in *front*. □

LEMMA 5.3.8. *BBFC(f + 1)-Termination: Every round eventually terminates.*

For lack of space, the proof of this lemma, which is technical, is deferred to the full version of this paper [26].

DEFINITION 5.3.3. *Let r be a round of the algorithm, we define by $front_r = \{p | r_p > r + f + 1\}$ the group of nodes whose current round is greater by at least $f+1$ than r.*

LEMMA 5.3.9. *BBFC(f+1)-Finality: In every round $r' > r + f + 1$, v_p^r is definite.*

PROOF. Following Algorithm 2's code, after a valid block is WRB-delivered, the only way it can be replaced is by an invocation of the recovery procedure. By Algorithm 2's code and Lemma 5.3.7, if the recovery procedure has been invoked regarding round r, then p adopts a prefix version that was suggested by $q \in$ *front*. Obviously, *front* $\subseteq front_r$. By Lemma 5.3.5, $\forall q, p \in front_r, b_q^r = b_p^r$, i.e., b_p^r is definite.

Note that the ratio between n and f as well as the version's validation test ensure that no Byzantine node is able to propose a version that does not include b_q^r. □

From Lemmas 5.3.1, 5.3.5, 5.3.8 and 5.3.9 we have:

THEOREM 5.3.1. *Algorithm 2 solves BBFC(f + 1).*

	fault-free	Timing, and omission failures	Byzantine failures
Communication steps	1	$2 + OBBC$	RB $+n$ parallel AB
Digital signature operations	1	$OBBC$	RB+AB $+(n-f)$ ·(chain size)
Latency (in rounds)	$f+1$	no additional overhead	no additional overhead

Table 1: Performance characteristics of FireLedger. RB and AB stand for Reliable and Atomic Broadcast respectively. The first column shows the performance in the fault-free synchronized case. The second column depicts the additional costs in an unsynchronized period and omission failures. The third column depicts the additional expenses in the presence of a non-benign fault.

LEMMA 5.3.10 (ATOMIC ORDER). *Correct nodes using Algorithm 2 deliver blocks in the same order.*

PROOF. Follows directly from Lemma 5.3.5 and the iterative nature of the algorithm. □

By Lemma 5.3.10 and Theorem 5.3.1, we have:

THEOREM 5.3.2. *The protocol listed in Algorithm 2 impose a total order of all blocks.*

THEOREM 5.3.3. *The protocol listed in Algorithm 2 satisfies Non-Triviality.*

The proof of Theorem 5.3.3 follows directly from Lemmas 5.3.8 and 5.3.9, given that correct nodes propose non-empty blocks whenever they hold clients' transactions that have not been included in any previously decided block.

5.4 Theoretical Bounds and Performance

Table 1 summarizes the performance of FireLedger in each of its three modes. In case of no failures and synchronized network, FireLedger performs in the OBBC of WRB-deliver a single all-to-all communication step as well as one digital signature operation. Further, only one node broadcasts more than one bit. This can be obtained since the latency for a definitive decision can be up to $f + 1$ rounds. In case a message is not received by WRB-deliver due to timing, omission or benign failures, the algorithm runs the non optimistic phase of OBBC as well as two more communication steps (one-to-all and one-to-one) in which a node asks for the missed message from the nodes who did receive it. Also, the amount of digital signature operations depends on the specific OBBC implementation that is used by WRB-deliver. Finally, in case of Byzantine failures, FireLedger runs the recovery procedure which depends on the Reliable Broadcast and the Atomic Broadcast implementations. Yet, when Byzantine failures manifest, a node does not lose blocks so the latency in terms of rounds remains the same.

6. IMPLEMENTATION

6.1 Optimizations

Separating Headers and Blocks. FireLedger's protocol enables to easily separate the data path from the consensus path, such that only block headers need to pass through

the consensus layer while the block itself is being sent asynchronously in the background.

That is, a node p broadcasts a block as soon as the block is ready. On p's next proposing round, p WRB-broadcasts a header of a previously sent block. Respectively, upon WRB-delivering a header, if p did not receive the block, it votes against delivering it (See Algorithm 1, lines 8–13). In addition, if a decision was made to deliver a header, but the block itself has not been received by p, then p has to retrieve the block from a correct node q that has it. Such a q exists because the decision to deliver is done only if at least one correct node has voted in favor of delivering, which means that q has the block.

Dynamically Tuning the Timeout. To adjust the timer to the current network delays status, we dynamically adjust its value based on the exponential moving average (EMA) of the message delays over the last N rounds. Namely, denote by $d_k, timer_k$ the delay of a message and the timer of round k respectively. Then for every round r

$$timer_r = \frac{2}{N+1} \cdot d_{r-1} + timer_{r-2} \cdot (1 - \frac{2}{N+1}).$$

A formal discussion of the above tuning model is out of the scope of this paper, but see, e.g., [19, 34, 47].

Benign FD. FireLedger's algorithm enables implementing a simple benign failure detector (FD) such that a crashed node will not cause an unrestricted increase in the timer value. Largely speaking, every node p maintains a suspected list of the f nodes to which p has waited the most and above a predefined threshold. For every node q in that list, on a WRB-deliver, p does not wait for q's message but rather immediately votes against delivering. By the ratio between n and f there is at least one correct node c that is not suspected by any correct node and thus the algorithm's liveness still holds. Despite the above, if c is one of the last f proposers it would not be able to suggest a new block. Hence, the suspected list is invalidated every time FireLedger is skipping a node that is in the last f proposers (see Algorithm 2, lines 1–3). Also, if a Byzantine activity was detected, to avoid considering more than f nodes as faulty, we invalidate the suspected list.

6.2 FireLedger's Instance Implementation

Figure 2 depicts the main components of a FireLedger's instance. Using FireLedger's API one can feed the *TX pool* with a new write request. The *main thread* creates a new block and *WRB-broadcasts* it in its turn. In addition, the *main thread* tries to *WRB-deliver* blocks relying on *OBBC*. If it succeeds, the block is added to the *Blockchain*. Meanwhile, the *panic thread* waits for a panic message. When such a message is *Atomic-delivered*, the *panic thread* interrupts the *main thread* which as a result invokes the *recovery* procedure. FireLedger is implemented in Java and the communication infrastructure uses gRPC, excluding BFT-SMaRt which has its own communication infrastructure. *Atomic Broadcast* is natively implemented on top of *BFT-SMaRt* whereas *OBBC* uses *BFT-SMaRt* only as the fallback mechanism when agreement cannot be reached through the optimistic fast path (which relies on gRPC).

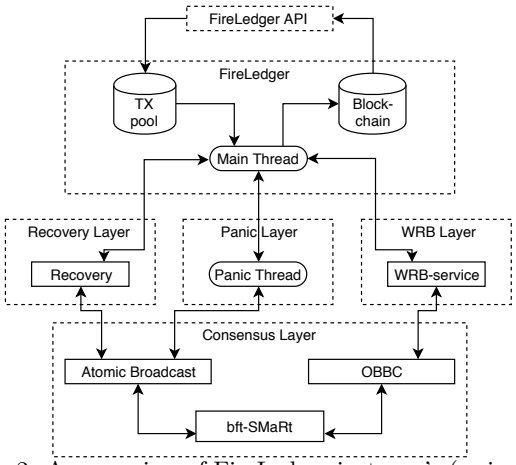

Figure 2: An overview of FireLedger instance's (a single FL worker) main components

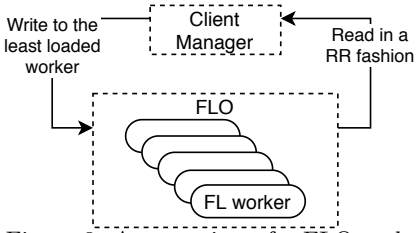

Figure 3: An overview of a FLO node

6.3 FLO – FireLedger Orchestrator

While FireLedger assumes partial synchrony, its rotating leader pattern imposes synchronization in that a node may propose a value only on its turn, making FireLedger's throughput bounded by the actual network's latency. To ameliorate this problem, we introduce another level of abstraction, named *workers*, by which each node runs multiple instances of FireLedger and uses them as a blockchain based ordering service. The use of workers brings two benefits: (*i*) workers behave asynchronously to each other which compensates for the above synchrony effect of FireLedger and (*ii*) while a worker waits for a message, other workers are able to run, resulting in better CPU utilization. To preserve the overall total ordering property of FireLedger, a node must collect the results from its workers in a pre-defined order, e.g., round robin. This requirement may imposes higher latency when the system is heavily loaded because even if a single worker faces the non-optimistic case, it delays all other workers from delivering their blocks to the node.

Figure 3 depicts an overview of a FLO node. Upon receiving a write request, the *client manager* directs the request to the least loaded worker. When a read request is received, the *client manager* tries to read the answer from the relevant worker. Only if the answer was already definitely decided and its block can be delivered in the pre-defined order, the node returns the answer[4].

An alternative design could be to maintain a semi-static leader that only changes when it misbehaves, while enabling it to propose new blocks back-to-back up to some threshold window. Such a design offers another performance tradeoff between latency and throughput compared to FLO. We have

[4]This is similar to the *deterministic merge* idea of [14].

parameter	range	units
cluster size	$n \in \{4, 7, 10\}$	-
workers	$1 \leq \omega \leq 10$	-
transaction size	$\sigma \in \{512, 1K, 4K\}$	Byte
batch size	$\beta \in \{10, 100, 1000\}$	Transaction

Table 2: The default evaluation's parameters: The first row presents FLO's cluster size. The system model assumes $f < \frac{n}{3}$, hence, $n \in \{4, 7, 10\}$ imposes $f \in \{1, 2, 3\}$ respectively.

chosen FLO since it fits more naturally to the overall design of FireLedgerand its rotating leader paradigm[15, 35, 73].

7. PERFORMANCE EVALUATION

Our evaluation studied the following questions: (*i*) Is FLO/FireLedger CPU bounded or network bounded? (*ii*) How do Table 2's values affect FireLedger's performance? (*iii*) How does node distribution affect FireLedger's performance? (*iv*) How does FireLedger handle failures?

Deployment Specification. Our setup for most measurements includes n nodes running on n identical VMs with the following specification: m5.xlarge with 4 vCPUs of Intel Xeon Platinum 8175 2.5 GHz processor, 16 GiB memory and up to 10 Gbps network links (Section 7.6 uses a stronger configuration as detailed there).

Workload Specification. To stress test FireLedger's performance as an ordering protocols for blockchains, we have used the following workloads: During every run, each proposer in its turn generates a block containing β transactions of σ bytes each. All transactions are "write" transactions, meaning that they are ordered by the protocol. We do not simulate the transactions execution time, but rather after a block enters the blockchain, it is asynchronously written to the disk and database. To reduce variability caused by external factors, we have measured the throughput as experienced by each node (i.e., total definite transactions during the test time) and the latency as the time from when the proposer generates the block until it becomes definite. Note that we create new random transactions for every new proposal, meaning that we have to calculate new signatures and hashes for each new block.

7.1 Signature Generation

In FLO/FireLedger, as long as the optimistic assumptions hold, a proposer signs its block only once and any other node is verifying the signature only once. Hence, the maximal signature rate serves as an upper bound on the potential throughput of FireLedger.

To undrestand whether FLO/FireLedger is CPU bounded or I/O bounded we start by presenting an evaluation of the signatures generation rate (sps) which is typically the most CPU intensive task. We use ECDSA signatures with the *secp256k1* curve. When signing a block, all the block's transactions are hashed and the result is signed alongside the block header. We vary the ω, β and σ values in the ranges described in Table 2.

For each configuration we run the benchmark for 1 minute. Figure 4 depicts the benchmark results. As expected, with small blocks the sps is higher than with larger ones. Also, as our machines have 4 vCPUs, increasing ω beyond 4 has

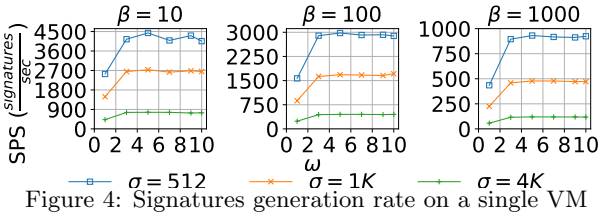

Figure 4: Signatures generation rate on a single VM

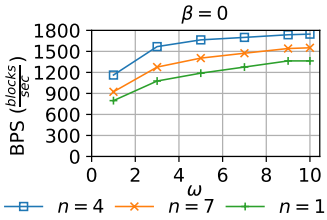

Figure 5: FLO's bps rate for $n \in \{4, 7, 10\}$ in a single data-center cluster

a minor effect if any. As seen below, the performance of FLO is not limited by the *sps* rate.

7.2 FLO Cluster in a Single Data-Center

We deployed a FLO cluster where all machines reside in the same data center. In the current version, FireLedger uses a clique overlay for disseminating both blocks and headers. To avoid clogging the network, FireLedger has a basic flow control mechanism that prevents nodes from sending new blocks if the network is overloaded or if they have already disseminated enough blocks that have not been decided yet. Due to its modular design, a more sophisticated mechanism can be plugged into the system. This is left for future work.

7.2.1 FLO's Throughput

For each configuration we run the experiment for 3 minutes. The results were collected from all nodes and we took the average among them.

7.2.1.1 BPS Rate.

Due to the separation of blocks and headers, FLO's throughput is mostly bounded by its bps rate. Figure 5 presents FLO's bps rate for different n and ω values. As expected, increasing ω yields higher bps due to better CPU utilization. In contrast, increasing n decreases the bps because each decision requires more communication. Even so, FLO delivers thousands of bps under the majority of the tested configurations. FLO's throughput is bounded by $tps \leq \beta \cdot bps$.

7.2.1.2 TPS Rate.

We tested FLO's throughput while varying n, ω, σ and β values in the ranges described in Table 2. Figure 6 shows FLO's throughput with the above configurations. Except few configurations where $\beta = 10$, FLO achieves between tens to hundreds of thousands of tps, depending on the specific configuration. Especially, with $\sigma = 512$, which according to [11] is the average size of a Bitcoin's transaction, FLO peaks at around 160K tps even with $n = 10$. As expected, for larger σ the performance decreases because less of the network's bandwidth remains available for the headers, which limits the bps rate. It can be observed that the performance for large blocks with $n \in \{7, 10\}$ is better than

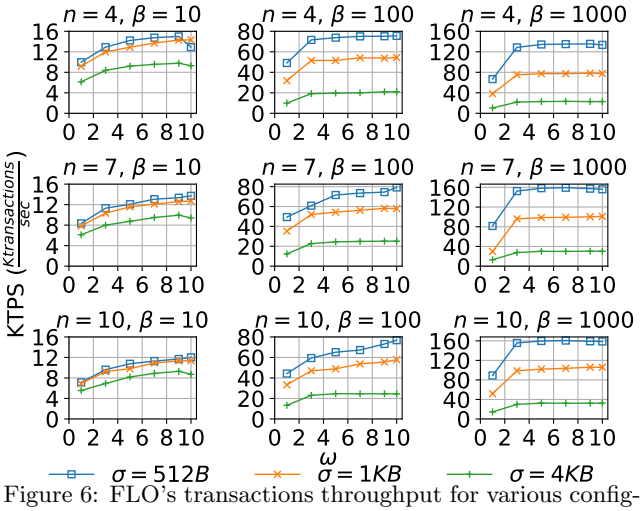

Figure 6: FLO's transactions throughput for various configurations in a single data-center deployment

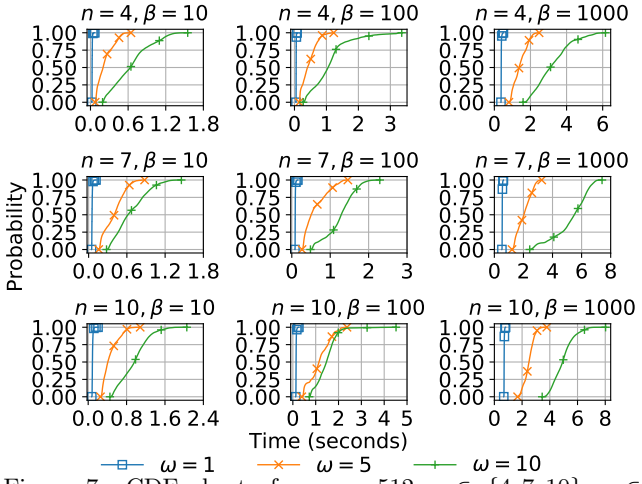

Figure 7: CDF charts for $\sigma = 512, n \in \{4, 7, 10\}, \omega \in \{1, 5, 10\}$ and $\beta \in \{10, 100, 1000\}$ in a single data-center

when $n = 4$. This can be explained by the fact that the separation of blocks from headers allows for nodes in bigger clusters to collect more blocks that have not been decided yet. Thus, as the bps grows w.r.t. the number of workers, it respectively increases the tps rate. This does not manifest in small block sizes because with very small blocks there is little benefit from transmitting a block before its header. Further, when $\omega = 10$, there are significantly more workers than hardware threads, which increases performance sensitivity sometimes yielding lower overall throughput.

7.2.2 *FLO's Latency*

To evaluate FLO's latency, we measured the time it takes for a full block to be delivered by FLO. This time includes disseminating the block, its header, and to wait until it can be delivered by the round robin between FLO's workers. We focus on the configurations in which $\sigma = 512$ (same as Bitcoin transactions). Figure 7 shows CDF charts for various configurations. As expected, with $\omega = 1$ FLO's latency is minimal and is less than 1 second even with $n = 10$ and $\beta = 1000$. Increasing ω results in an increase in the latency respectively. This is due to the fact that even a single

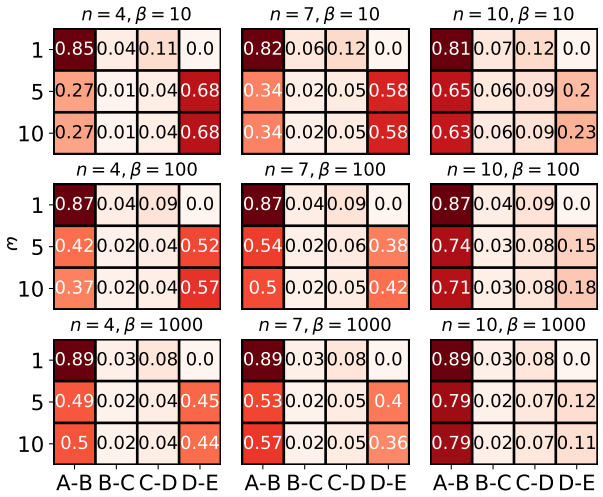

Figure 8: Heatmaps that depicts the relative execution time of FLO between 5 different events for $\sigma = 512$

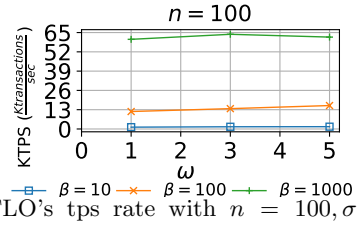

Figure 9: FLO's tps rate with $n = 100, \sigma = 512, \beta \in \{10, 100, 1000\}$ and $1 \leq \omega \leq 5$

worker's delay is reflected in all, due to FLO's round robin. Yet, even with $n = 10, \omega = 10$ and $\beta = 1000$ the latency in a single data-center deployment is below 8 seconds.

To understand the main bottlenecks of FLO, we divided every round of the algorithm into 5 different events: (A) block proposal, (B) header proposal, (C) tentative decision, (D) definite decision, and (E) delivering by FLO. Finally, we measured the time between each pair of consecutive events.

Figure 8 shows the *relative* execution time between each two consecutive events. It is easy to see that due to the separation of blocks and headers, the majority of time is spent between receiving a block and receiving its header. In addition, for $\omega > 1$, the workers cause an increase in the latency, as even a single worker's delay delays the whole system. Finally, increasing n as well as β causes the blocks dissemination event to take longer. This is despite using a clique layout. Other methods (e.g., gossip) may improve the throughput but not the latency.

7.3 FLO's Scalability

To test FLO's scalability we deployed a single data-center cluster of $n = 100$ machines and tested FLO's tps rate with $\sigma = 512, \beta \in \{10, 100, 1000\}$ and $1 \leq \omega \leq 5$. We ran each configuration for 3 minutes. Figure 9 depicts the benchmark's result. Thanks to FireLedger's frugal communication pattern, as long as the fault free execution path take place, FLO can achieve around $60K$ tps (in a single data-center deployment). As shown, in this cluster size, the number of workers has no effect because of the relatively large amount of communication that even a single worker consumes.

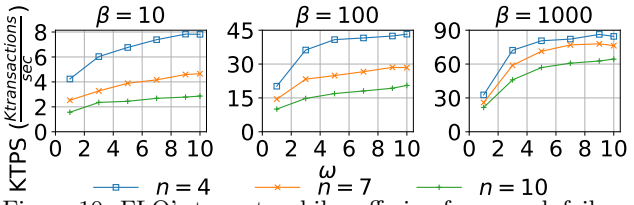

Figure 10: FLO's tps rate while suffering from crash failures of f nodes for $\sigma = 512, \beta \in \{10, 100, 1000\}, n \in \{4, 7, 10\}$ and $f \in \{1, 2, 3\}$.

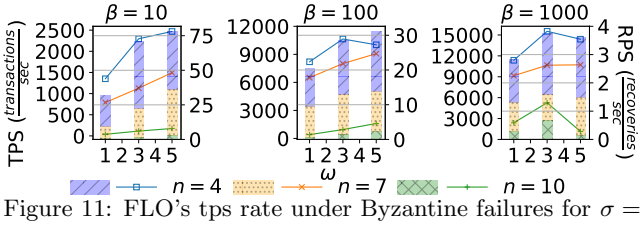

Figure 11: FLO's tps rate under Byzantine failures for $\sigma = 512, 1 \leq \omega \leq 5, \beta \in \{10, 100, 1000\}, n \in \{4, 7, 10\}$ and $f \in \{1, 2, 3\}$. The lines show the tps rate and the bars shows the recovery per second (rps) rate

7.4 FLO Under Failures

7.4.1 Benign Failures

We tested FLO while suffering from crash failures of f nodes (yet, we maintain $n = 3f+1$ even in this benign case). Here, all faulty nodes crash in the middle of a run (such a node crashes with all of its workers), but the measurements are taken after the faulty nodes crash. Every benchmark was ran for 3 minutes and we calculated the average tps among the correct nodes. Figure 10 depicts FLO's tps rate under various configurations while facing crash failures.

It can be seen that due to the full BBC phase that is now needed during faulty nodes rounds, for larger n the tps is decreasing. Yet, FLO still reaches tens of thousands of tps despite these benign failures. This is due to the OBBC protocol and the basic failure detector described in Section 6.1.

7.4.2 Byzantine Failures

To test FLO's performance when facing Byzantine failures, we deployed a Byzantine FLO node that operates as following: When started, every worker divides the cluster into two random parts, and for every given round it distributes different versions of the block to each part. Notice that in practice, invoking the recovery procedure may cause the nodes to become un-synchronized with each other, a fact that affects the performance as well. This increases the variance between measurements of the same settings. Hence, to be able to perform more measurements of each data point, for each configuration we run a series of short benchmarks (between 1 - 2 minutes each).

Figure 11 presents the tps rate for FLO when facing Byzantine failures w.r.t. the number of workers and the number of recoveries per second (rps). Smaller values of β and n imply more recovery events. Recall that during the recovery nodes halt. Thus, the above is expected due to the fact that each recovery ends faster when β and n are small. Yet, for bigger β, the batching effect compensate for the small amount of recoveries and the long halts. The reason why for $n = 10, \beta = 1000$ and $\omega = 5$ the performance decreased

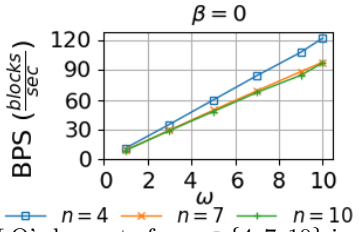

Figure 12: FLO's bps rate for $n \in \{4, 7, 10\}$ in a multi data-center cluster

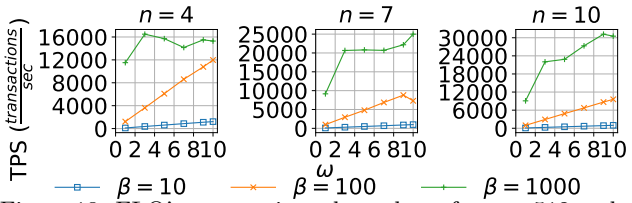

Figure 13: FLO's transactions throughput for $\sigma = 512$ under various configurations in a multi data-center

so much is the underlying Byzantine consensus layer (BFT-SMaRt), which has to handle a large amount of data. To conclude, FLO delivers more than 10K tps in some scenarios even when facing Byzantine failures. Although the performance is lower than in optimistic executions, these type of failures are expected to be rare in permissioned blockchain clusters. And even with $n = 10$, if we set $\beta = 1000$ and $\omega = 3$ we obtain about 6K tps, roughly twice the average tps of VISA. Hence, FLO presents an attractive trade-off between scalability, performance and security.

7.5 FLO in a Multi Data-Center Cluster

We also tested FLO in a geo-distributed setting with nodes spread around the world. The nodes were placed, one node per region, by the following order, in Amazon's Tokyo, Central, Frankfurt, Paris, Sau-Paulo, Oregon, Singapore, Sydney, Ireland and Ohio data-centers. We tested only fault free scenarios. Thus, we kept using BFT-SMaRt rather than its geo-distributed optimized version named WHEAT [71].

7.5.1 FLO's Throughput

As before, we first measured the bps rate for $n \in \{4, 7, 10\}$. Figure 12 depicts the bps rate for varying cluster sizes. As expected, due to lower network performance, the bps is less than 10% of its rate in single data-center clusters.

As in the single data-center deployment, we simulated high load by creating random transactions and run FLO with the following configurations: $n \in \{4, 7, 10\}, 1 \leq \omega \leq 10, \sigma = 512, \beta \in \{10, 100, 1000\}$. Every benchmark was run for 3 minutes. Figure 13 presents the benchmark's result. Obviously, tps is increasing with w and β due to a better CPU utilization and a batching effect. As for the increase of the tps with n, this is, as before, thanks to the separation of blocks from headers, which allows bigger clusters to collect more blocks to decide on, thereby enhancing performance.

7.5.2 FLO's Latency

We tested FLO's latency in the above multi data-center deployment. As before, we measured the time that takes a block to be delivered by FLO from the moment it was proposed by its creator. To avoid outliers, we omitted the 5%

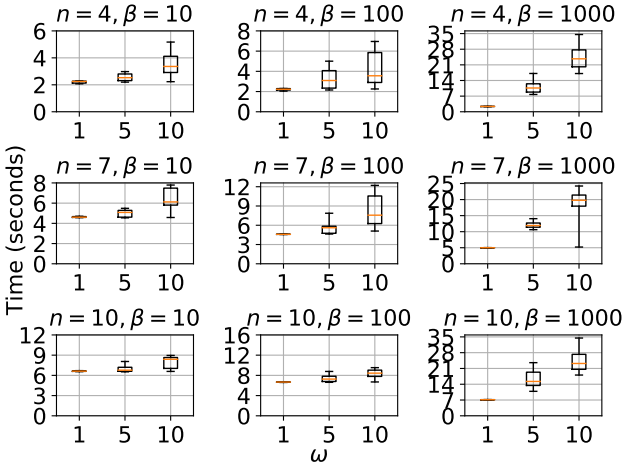

Figure 14: FLO's latency in a multi data-center deployment for $\sigma = 512, \omega \in \{1, 5, 10\}, \beta \in \{10, 100, 1000\}$ and $n \in \{4, 7, 10\}$

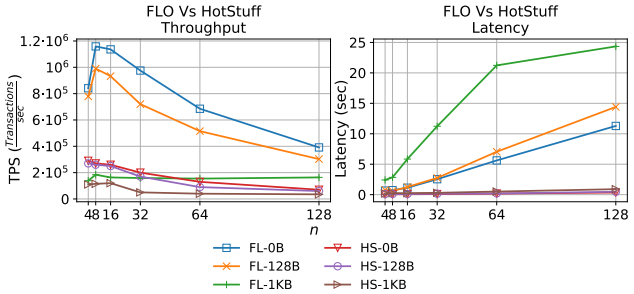

Figure 15: Comparison of FLO and HotStuff on c5.4xlarge AWS machines and $f = \lceil n/3 \rceil - 1$

most extreme results. Figure 14 presents the benchmarks results. For small blocks, the cluster size slightly affects the latency because the headers dissemination process dominates the bandwidth. Yet, with large blocks, as the data dissemination process itself dominates the latency, increasing the cluster size has very little effect on the latency.

7.6 FireLedger vs. Leading Alternatives

To the best of our knowledge, the current best performing alternative to FireLedger is HotStuff [75]. As we had no access to the codebase of HotStuff, we compare the declared performance of HotStuff from [75] with our own measurements of FireLedger using the exact same environment as in [75], namely c5.4xlarge AWS machines (16 vCPUs, 32 GiB RAM). We also compare with the numbers obtained for BFT-SMaRt [20], the previous state-of-the-art system, in this same setting. Figure 15 and 16 show this comparison's results w.r.t the n and σ parameters. For all cluster sizes FLO was deployed with $\beta = 1000$ and $\omega = 8$ with maximal resiliency $f = \lceil n/3 \rceil - 1$. In terms of throughput, for any σ and n FLO performs $20\% - 300\%$ better than HotStuff and $40\% - 600\%$ better than bft-SMaRt. Notice that HotStuff is implemented in C while FLO is in Java.

This performance gap is due to the fact that in the optimistic case, HotStuff requires all nodes to sign an asymmetric signature on each block and the proposer to verify $n - 1$ such signatures. In contrast, in FireLedger only the proposer that generates the block signs an asymmetric signature. Since signing is a heavy pure CPU time operation, fewer asymmetric signatures enable better throughput.

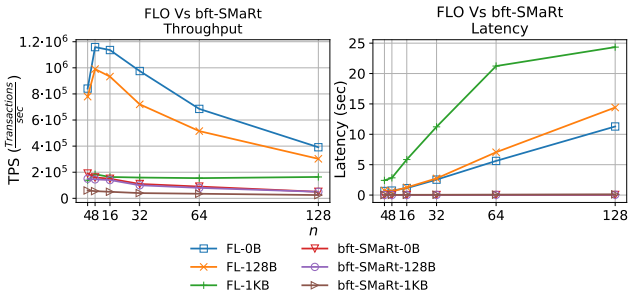

Figure 16: Comparison of FLO and bft-SMaRt on c5.4xlarge AWS machines and $f = \lceil n/3 \rceil - 1$

As for latency, due to the $f + 1$ finality of FireLedger and the fact that we run with maximal resiliency, FLO's latency rises with n. In contrast, transactions' finality with both HotStuff and bft-SMaRt is at most three rounds (HotStuff), so their latency is much less impacted by n. Still, in all cases the latency obtained by FLO is better than most existing cryptocurrencies (and tokens), including the very recent Algo (Algorand) [51]. Libra's target 10 seconds finality [45] is met by FLO when $n \leq 30$ nodes[5].

Conclusions. The performance gap between FLO and the alternatives narrows as transactions become larger. This is because in such cases, the basic need to disseminate the transactions dominates the communication overhead, so a clever consensus protocol has less room for impact. Hence, one should consider compressing the data for large transactions. Also, by employing scalable Byzantine dissemination protocols for the transactions data, e.g., [40], FLO is likely to better handle large clusters.

8. DISCUSSION

FireLedger is a communication frugal optimistic blockchain algorithm targeting environments where failures rarely occur. For example, FireLedger is likely to be very attractive for the FinTech industry, which uses highly secure and robust systems. FireLedger leverages blockchain's iterativity as well as its cryptographic features to achieve its goal.

Intuitively, FLO employs FireLedger as a blockchain-based consensus algorithm rather than consensus-based blockchain. Our performance results show that it matches the requirements of real demanding commercial applications even when executing on common non-dedicated infrastructure. Our design is especially suitable for cryptocurrencies and financial applications that tolerate a few seconds latency. In the future, we intend to explore sharding, which can potentially give an additional significant performance boost, as well as scalable dissemination protocols. Finally, our prototype implementation of this work is available in open source [10].

Acknowledgments. We would like to thank the anonymous reviewers for their many insightful comments. This work was partially funded by ISF grant #1505/16.

[5]Ripple and Stellar, for example, run on smaller clusters.

9. REFERENCES

[1] Chain core. https://chain.com/.

[2] Corda. https://github.com/corda/corda.

[3] fabric-orderingservice. https://github.com/bft-smart/fabric-orderingservice.

[4] Honeybadger bft. https://github.com/amiller/HoneyBadgerBFT.

[5] Hyperledger-fabric. https://github.com/hyperledger/fabric.

[6] Iroha. https://www.hyperledger.org/projects/iroha.

[7] Red belly blockchain. https://redbellyblockchain.io.

[8] Symboint-assembly. https://symbiont.io/technology.

[9] Tendermint. https://github.com/tendermint/tendermint.

[10] TOP/TOY Implementation. https://github.com/TopToy/TopToy.

[11] Tradeblock. https://tradeblock.com/bitcoin/historical/1w-f-tsize_per_avg-11101.

[12] PwC's Global Blockchain Survey. https://www.pwccn.com/en/research-and-insights/publications/global-blockchain-survey-2018/global-blockchain-survey-2018-report.pdf, 2018.

[13] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine Fault-tolerant Services. *ACM SIGOPS Oper. Syst. Rev.*, 39(5):59–74, Oct. 2005.

[14] M. K. Aguilera and R. E. Strom. Efficient Atomic Broadcast Using Deterministic Merge. In *Proc. of the 19th Annual ACM Symposium on Principles of Distributed Computing*, PODC, pages 209–218, 2000.

[15] Y. Amir, B. A. Coan, J. Kirsch, and J. Lane. Prime: Byzantine Replication under Attack. *IEEE Trans. Dependable Sec. Comput.*, 8(4):564–577, 2011.

[16] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. W. Cocco, and J. Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. *CoRR*, abs/1801.10228, 2018.

[17] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. John Wiley Interscience, March 2004.

[18] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The Next 700 BFT Protocols. *ACM Trans. Comput. Syst.*, 32(4), Jan. 2015.

[19] M. Bertier, O. Marin, and P. Sens. Implementation and Performance Evaluation of an Adaptable Failure Detector. In *Proc. of Int. Conf. on Dependable Systems and Networks (DSN)*, pages 354–363, June 2002.

[20] A. Bessani, J. Sousa, and E. E. P. Alchieri. State Machine Replication for the Masses with BFT-SMART. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 355–362, June 2014.

[21] K. Birman and R. Friedman. Trading Consistency for Availability in Distributed Systems. Technical Report TR96-1579, Computer Science Department, Cornell University, Apr. 1996.

[22] D. Boneh, B. Lynn, and H. Shacham. Short Signatures from the Weil Pairing. *Journal of Cryptology*, 17(4):297–319, Sep 2004.

[23] Z. Bouzid, A. Mostfaoui, and M. Raynal. Minimal Synchrony for Byzantine Consensus. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC, pages 461–470, 2015.

[24] G. Bracha. Asynchronous Byzantine Agreement Protocols. *Information and Computation*, 75(2):130–143, 1987.

[25] E. Buchman. Tendermint: Byzantine Fault Tolerance in the Age of Blockchains. Master's thesis, University of Guelph, 2016.

[26] Y. Buchnik and R. Friedman. FireLedger: A High Throughput Blockchain Consensus Protocol. *CoRR*, abs/1901.03279, November 2019. Full Version.

[27] Y. Buchnik and R. Friedman. A Generic Efficient Biased Optimizer for Consensus Protocols. In *21st International Conference on Distributed Computing and Networking (ICDCN)*, 2020.

[28] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and Efficient Asynchronous Broadcast Protocols. In *Advances in Cryptology – CRYPTO*, pages 524–541. Springer Berlin Heidelberg, 2001.

[29] C. Cachin and M. Vukolic. Blockchain Consensus Protocols in the Wild. *CoRR*, abs/1707.01873, 2017.

[30] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the 3rd ACM Symposium on Operating Systems Design and Implementation*, OSDI, pages 173–186, 1999.

[31] Chain. Federated-Consensus White Paper. Available online, https://chain.com/docs/1.2/protocol/papers/federated-consensus.

[32] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[33] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[34] W. Chen, S. Toueg, and M. K. Aguilera. On the Quality of Service of Failure Detectors. *IEEE Transactions on Computers*, 51(1):13–32, Jan 2002.

[35] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, pages 153–168, 2009.

[36] M. Correia, N. F. Neves, and P. Veríssimo. From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols Without Signatures. *Comput. J.*, 49(1):82–96, Jan. 2006.

[37] T. Crain, V. Gramoli, M. Larrea, and M. Raynal. DBFT: Efficient Leaderless Byzantine Consensus and its Application to Blockchains. In *17th IEEE International Symposium on Network Computing and Applications (NCA)*, 2018.

[38] F. Cristian, H. Aghili, H. R. Strong, and D. Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. *Inf. Comput.*, 118(1):158–179, 1995.

[39] D. Dolev, C. Dwork, and L. Stockmeyer. On the Minimal Synchronism Needed for Distributed Consensus. *J. ACM*, 34(1):77–97, Jan. 1987.

[40] V. Drabkin, R. Friedman, G. Kliot, and M. Segal. RAPID: Reliable Probabilistic Dissemination in Wireless Ad-Hoc Networks. In *26th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 13–22, 2007.

[41] S. Duan, H. Meling, S. Peisert, and H. Zhang. BChain: Byzantine Replication with High Throughput and Embedded Reconfiguration. In M. K. Aguilera, L. Querzoni, and M. Shapiro, editors, *Principles of Distributed Systems*, pages 91–106. Springer International Publishing, 2014.

[42] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *J. ACM*, 35(2):288–323, Apr. 1988.

[43] M. Ebrahim, S. Khan, and U. Khalid. Symmetric Algorithm Survey: A Comparative Analysis. *CoRR*, abs/1405.0398, 2014.

[44] M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, and R. Ramamurthy. Blockchaindb: A shared database on blockchains. *Journal of VLDB Endowment*, 12(11):1597–1609, July 2019.

[45] Z. A. et al. The Libra Blockchain. 2019.

[46] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, Apr. 1985.

[47] F. C. Freiling, R. Guerraoui, and P. Kuznetsov. The Failure Detector Abstraction. *ACM Computing Surveys*, 43(2), Feb 2011.

[48] R. Friedman, A. Mostefaoui, and M. Raynal. Simple and efficient oracle-based consensus protocols for asynchronous byzantine systems. *IEEE Trans. Dependable Secur. Comput.*, 2(1):46–56, Jan. 2005.

[49] R. Friedman and R. van Renesse. Packing Messages as a Tool for Boosting the Performance of Total Ordering Protocols. In *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 233–242, 1997.

[50] J. Garay, A. Kiayias, and N. Leonardos. The Bitcoin Backbone Protocol: Analysis and Applications. In *Advances in Cryptology – EUROCRYPT*, pages 281–310. Springer Berlin Heidelberg, 2015.

[51] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, SOSP, pages 51–68, 2017.

[52] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma. Throughput Optimal Total Order Broadcast for Cluster Environments. *ACM Trans. Comput. Syst.*, 28(2):5:1–5:32, July 2010.

[53] M. Hurfin, A. Mostefaoui, and M. Raynal. A Versatile Family of Consensus Protocols Based on Chandra-Toueg's Unreliable Failure Detectors. *IEEE Transactions on Computers*, 51(4):395–408, April 2002.

[54] Z. István, A. Sorniotti, and M. Vukolić. Streamchain: Do blockchains need blocks? In *Proceedings of the 2Nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, SERIAL, pages 1–6. ACM, 2018.

[55] D. Johnson, A. Menezes, and S. Vanstone. The Elliptic Curve Digital Signature Algorithm (ECDSA). *International Journal of Information Security*, 1(1):36–63, Aug 2001.

[56] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. *SIGOPS Oper. Syst. Rev.*, 41(6):45–58, Oct. 2007.

[57] K. Kursawe. Optimistic Byzantine Agreement. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 262–267, 2002.

[58] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[59] L. Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[60] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[61] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena. A Secure Sharding Protocol For Open Blockchains. In *Proc. of the ACM SIGSAC Conference on Computer and Communications Security*, CCS, pages 17–30, 2016.

[62] J.-P. Martin and L. Alvisi. Fast Byzantine Consensus. *IEEE Trans. Dependable Secur. Comput.*, 3(3):202–215, July 2006.

[63] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The Honey Badger of BFT Protocols. In *Proc. of the ACM SIGSAC Conference on Computer and Communications Security*, CCS, pages 31–42, 2016.

[64] A. Mostèfaoui, H. Moumen, and M. Raynal. Signature-free asynchronous binary byzantine consensus with t≤n/3, o(n2) messages, and o(1) expected time. *J. ACM*, 62(4), Sept. 2015.

[65] D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the USENIX Annual Technical Conference*, ATC, pages 305–320, 2014.

[66] R. Pass and E. Shi. Thunderella: Blockchains with Optimistic Instant Confirmation. In J. B. Nielsen and V. Rijmen, editors, *Advances in Cryptology (EUROCRYPT)*, pages 3–33. Springer International Publishing, 2018.

[67] M. Raynal. *Fault-Tolerant Message-Passing Distributed Systems*. Springer International Publishing, 2018.

[68] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Commun. ACM*, 21(2):120–126, Feb. 1978.

[69] R. Roth. *Introduction to Coding Theory*. Cambridge University Press, 2006.

[70] F. B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[71] J. Sousa and A. Bessani. Separating the WHEAT

from the Chaff: An Empirical Design for Geo-Replicated State Machines. In *Proc. of IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*, pages 146–155, 2015.

[72] J. Sousa, A. Bessani, and M. Vukolic. A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform. *CoRR*, abs/1709.06921, 2017.

[73] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary. In *Proceedings of the 28th IEEE International Symposium on Reliable Distributed Systems*, SRDS, pages 135–144, 2009.

[74] M. Vukolic. The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication. In J. Camenisch and D. Kesdogan, editors, *iNetSeC*, volume 9591 of *Lecture Notes in Computer Science*, pages 112–125. Springer, 2015.

[75] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. HotStuff: BFT Consensus in the Lens of Blockchain. *CoRR*, abs/1803.05069, March 2019.