FairLedger: A Fair Blockchain Protocol for Financial Institutions

Kfir Lev-Ari
Technion IIT

Alexander Spiegelman[1]
VMware Research

Idit Keidar
Technion IIT

Dahlia Malkhi
Calibra

Abstract

Financial institutions nowadays are looking into technologies for permissioned blockchains. A major effort in this direction is Hyperledger, an open source project hosted by the Linux Foundation and backed by a consortium of over a hundred companies. A key component in permissioned blockchain protocols is a byzantine fault tolerant (BFT) consensus engine that orders transactions. However, currently available BFT solutions in Hyperledger (as well as in the literature at large) are inadequate for financial settings; they are not designed to ensure fairness or to tolerate the selfish behavior that inevitably arises when financial institutions strive to maximize their own profit.

We present FairLedger, a permissioned BFT blockchain protocol, which is fair, deigned to deal with rational behavior, and, no less important, easy to understand and implement. Our secret sauce is a new communication abstraction called *detectable all-to-all (DA2A)*, which allows us to detect players (byzantine or rational) that deviate from the protocol and punish them. We implement FairLedger in the Hyperledger open source project using the Iroha framework – one of the biggest projects therein. To evaluate FairLegder's performance, we also implement it in the PBFT framework and compare the two protocols. Our results show that in failure-free scenarios in wide-area settings, FairLedger achieves better throughput than both Iroha's implementation and PBFT.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Blockchain; Fairness; Byzantine fault tolerance; Rational players; Equilibrium

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.1

Related Version A full version of the paper is available at `https://arxiv.org/abs/1906.03819`.

1 Introduction

As of today, support for financial transactions between institutions is limited, slow, and costly. For example, an oversees money transfer between two banks might take several days and entail fees of tens of dollars. The source of this cost (in term of both time and money) is the need for a reliable clearing house; sometimes this even requires physical phone calls at the end of the day. At the same time, emerging decentralized cryptocurrencies like Bitcoin [42] complete transactions within less than hour, at a cost of microcents. It is therefore not surprising that financial institutions are looking into newer technologies to bring them up to speed and facilitate trading in today's global economy.

The most prominent technology considered in this context is that of a *blockchain*, which implements a secure peer-to-peer *ledger* of financial transactions on top of a consensus engine.

[1] Kfir Lev-Ari and Alexander Spiegelman contributed equally to this work.

23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 1; pp. 1:1–1:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

A major effort in this direction is Hyperledger [28], an open-source project hosted by the Linux Foundation and backed by a consortium of more than a hundred companies. Unlike anonymous cryptocurrencies with open participation, in blockchains for financial institutions – also called *permissioned blockchains* – every participant is pre-known and certified, so that it has to be responsible for its actions in the real world. Permissioned blockchains [28, 40, 45] thus abandon the slow and energy-consuming proof-of-work paradigm of Bitcoin, and tend to go back to more traditional distributed consensus protocols. Because of the high stakes, malicious deviations from the protocol (due to bugs or attacks), rare as they might be, should never compromise the service. Such deviations are modeled as *byzantine* faults [34], and to deal with them, proposed solutions use *byzantine fault tolerant (BFT)* consensus protocols.

Yet we believe that dealing with byzantine failures is only a small part of what is required in permissioned blockchains. In fact, a break-in that causes a bank's software to behave maliciously is so unusual that it is a top news story and is investigated by the FBI. On the other hand, financial institutions always try to maximize their own profit, and would never use a system that discriminates against them. Moreover, they can be expected to selfishly deviate from the protocol whenever they can benefit from doing so. In particular, financial entities typically receive a fee for every transaction they append to the shared ledger, and can thus be expected to attempt to game the system in a way that maximizes the rate of their own transactions in the ledger. Such *rational* behavior, if not carefully considered, not only can discriminate against some entities, but may also compromise safety.

Thus, in the FinTec context, one faces a number of important challenges that were not always emphasized in previous BFT work: (1) *fairness* in terms of the opportunities each participant gets to append transactions to the ledger; (2) expected *rational behavior* by all players; and (3) *optimized failure-free performance* in wide-area setting, given that financial institutions are usually very secure and inter-institutional platforms would be deployed over a secure WAN. In addition, it is important to stress (4) *protocol simplicity*, because complex protocols are inherently bug-prone and easier to attack. In this work we develop *FairLedger*, a new permissioned BFT blockchain protocol for the Hyperledger framework, which addresses all of these challenges. Our protocol is fair, designed for rational participants, optimized for the failure-free case, simple to understand, and easy to implement. Specifically, we show that following the protocol is an equilibrium, and that when rational participants do follow the protocol, they all get perfectly fair shares of the ledger.

Given that byzantine failures are rare, our philosophy is to optimize for the *normal mode* when they do not occur (as also emphasized in some previous works, e.g., Zyzzyva [32]). For this mode, we design a simple protocol that provides high performance when all players are rational but not byzantine. Under byzantine failures, the normal mode protocol remains safe and fair, but may lose progress. Upon detecting that a rogue participant is attempting to prevent progress, we switch to the *alert mode*. At this point, it is expected that real-world authorities (such as the FBI or Interpol) will step in to investigate the break-in. But such an investigation may take days to complete, and in the time being, the service remains operational – albeit slower – using the alert mode protocol.

An important lesson learned from the deployment of Paxos-like protocols in real systems such as ZooKeeper [31] and etcd [19] is that systems will only be used if they are easy to understand, implement, and maintain. Like these systems, we follow the Vertical Paxos [4, 33] approach of using a fixed set of participants (sometimes called quorum) for sequencing transactions and reconfiguring this set upon failures. Specifically, we designate a *committee* consisting of all the participants who are interested in issuing transactions and have them run a *sequencing protocol* to order their transactions. A complementary *master* service monitors the committee's progress and initiates reconfiguration when needed. Including all interested players in the committee is instrumental for fairness – this way, all committee members benefit from sequencing batches that include transactions by all of them.

We assume a loosely synchronous model, where a master can use a coarse time bound (e.g., one minute) to detect lack of progress. This bound is only used for failure recovery, and does not otherwise affect performance. A key feature of our alert mode is that whenever participants deviate from the protocol in a way that jeopardizes progress, they are accurately detected and so can be removed from the committee. Unlike in other Hyperledger protocols [45], FairLedger never indicts correct participants, allowing the system to heal itself following attacks.

The sequencing protocol uses all-to-all exchange of signed messages among committee members. Since the committee includes all participants and all messages are signed, the protocol can ensure safety despite byzantine failures of almost any minority. Specifically, for f failures, our protocol is correct whenever the number of participants satisfies $n \geq 2f + 3$. The flip side is that it is enough for one participant to withhold a single message in order to prevent progress. Such a deviation from the protocol is tricky to detect as one participant can claim that it had sent a message to another, while the recipient claims that the message has not arrived. To deal with such deviations, we define a new communication abstraction, which we call *detectable all-to-all (DA2A)*. Besides the standard *broadcast* and *deliver* API, DA2A exposes a *detect* method that returns an accurate and complete set of deviating participants.

We implement FairLedger's sequencing protocol in Iroha [45], which is part of the Hyperledger [28] open-source project, and compare its performance to their implementation. Specifically, since Iroha's implementation is modular, we are able to replace their BFT consensus protocol, (which is based on [23]), with our sequencing protocol without changing other components (e.g., communication, cryptographic, and database libraries). Experiments over WAN emulation [48] show that FairLeadger outperforms Iroha's BFT protocol in the vast majority of the tested scenarios (both in normal mode and in alert mode).

Since the Iroha system consists of many components (e.g., GRPC [30] communication) that may induce overhead, we also implement FairLedger's sequencing protocol in the PBFT [17] framework, which provides a clean environment to evaluate raw consensus performance. Our results show that Fairledger's latency is better than PBFT's in both the normal and alert modes. Fairledger's throughput exceeds PBFT's in normal mode but is inferior to it in the alert mode, although PBFT's advantage diminishes as the system scale grows.

In summary, this paper makes the following contributions:

1. We define a fair distributed ledger abstraction for rational participants.
2. We define a detectable all-to-all (DA2A) abstraction as a building block for such ledgers.
3. We design FairLedger, the first BFT blockchain protocol that ensures strong fairness when all participants are rational. FairLedger is safe under byzantine failures of almost any minority, and detects and punishes deviating (byzantine and rational) participants. It is also simple to understand and implement.
4. We substitute Iroha, which is one of the Hyperledger's existing sequencing protocol, with FairLedger with improved performance. We also implement FairLedger's sequencing protocol in the PBFT framework; FairLedger outperforms PBFT in the normal mode but achieves slightly lower throughput in the alert mode.

2 Problem Definition and System Model

We consider a set of players, each representing a real-world financial entity, jointly attempting to agree on a shared *ledger* of financial transactions. Every player has an unbounded stream of transactions that it wants to append to the ledger and we assume that the player benefits from doing so. A principal goal for our service is *fairness*, that is, providing all entities with equal opportunities for appending transactions.

2.1 Byzantine and rational behavior

Traditional distributed systems are managed by a single organization, where deviation from the protocol – referred to as byzantine behavior – is explained as a bug or by the deviating entity being hacked, and only a small subset of the players are byzantine. In this work, however, we seek a protocol that coordinates among many organizations that trade with financial assets. We thus have to take into account that *every* entity may behave *rationally*, and deviate from the protocol if doing so increases its benefit.

To reason about such rational behavior we assume that each entity can be either *byzantine* or *rational* [5,36,41]. A rational entity has a known utility function that it tries to maximize and deviates from the protocol only if this increases its utility, whereas a byzantine entity can deviate arbitrarily from the protocol (e.g., crash, withhold messages, or send incorrect protocol messages), i.e., its utility function is unknown.

Our system involves two types of entities – *players* and *auditors*. Players (e.g., banks) propose transactions to append to the ledger, while auditors oversee the system. The same physical entity may be both a player and an auditor, but other entities (e.g., government central banks) may also act as auditors. There are initially n players and any number of auditors. The number of byzantine players is bounded by a known parameter f, where $n \geq 2f + 3$. At most a minority of the auditors can be byzantine.

In order to prove that a protocol is correct in our model, we need to show that following the protocol is an equilibrium for rational entities even in the presence of f byzantine faults.

2.2 Distributed fair ledger

A *ledger* is an abstract object that maintains a log (i.e., sequence) of *transactions* from some domain $\mathcal{T}$. It supports two operations with the following sequential specification: An *append*(t), $t \in \mathcal{T}$, changes the state of the log by appending t to its end. A *read*(l) operation returns the last l transactions in the log. The log is initially empty.

The *utility function of a rational player* is the ratio of transactions that it appends to the ledger, i.e., the number of transactions it appends to the ledger out of the total number of transactions in the ledger. Between two ledgers with the same ratio, the longer one is preferred. This models players who care about the overall system progress but care more about getting their fair share of it.

The *utility function of an auditor* is the committee size in case progress is being made, and 0 in case the system stalls. In other words, the auditors aim to ensure the system's overall health. In case an entity acts as an auditor and as a player, the auditor's utility is the dominating and the player's utility breaks ties.

We require '*strict fairness*. Intuitively, this means that for every player p_1 that follows the protocol, at any point when the log contains k transactions appended by p_1, the log does not contain more than $k + 1$ transactions appended by any other player. In the full paper [35] we formalize and extend this definition to allow differential quality of service, whereby different players are allocated different shares of the log and these shares may change over time.

2.3 System model

We assume that players have been certified by some trusted certification authority known to all players. In addition, we assume a PKI [44]: each player has a unique pair of public and private cryptographic keys, where the public keys are known to all players, and the adversary does not have enough computational power to unravel non-byzantine players' private keys.

We assume reliable communication channels between pairs of players. As in previous works on permissioned blockchains [23, 28, 45], we assume that there is a known upper bound Δ on message latency. Nevertheless, our sequencing protocol is safe and fair even if the bound does not hold. We exploit this bound to detect failures when the protocol stalls because a rogue player deviates from the protocol by withholding messages. Thus, the bound can be set very conservatively (e.g., in the order of minutes) so as to avoid false detection.

3 Solution Components

Our goal is to design a ledger that financial institutions will be able to use. Such a protocol, besides being fair, secure against malicious attacks, and resilient to selfish behavior, must be simple to understand, implement, and maintain. Therefore, although we appreciate complex protocols with many corner cases and clever optimizations, we try here to keep the design as simple as possible. The simple design not only reduces vulnerabilities, it also makes it much easier to reason about selfish behavior.

Committee and master. We adopt the Vertical Paxos [4, 33] paradigm, where a single committee (known to all) partakes in agreeing on all transactions. Initially, the committee consists of all players. By requiring all committee members to endorse transactions, we create an incentive for all of them to append to the log batches including transactions from all of them. To handle cases when committee members stop responding (e.g., due to a crash or an attack), a complementary *master* service performs *reconfiguration*: detecting such members and removing (or replacing) them. Thus, we logically implement two components: (1) a committee that runs the sequencing protocol and (2) a master responsible for progress. The master is implemented by auditors using a minority-resilient synchronous BFT protocol like [21]; its impact on overall system performance is small, and so we do not optimize its implementation. For the remainder of this paper, we abstract away this protocol and simply treat the master as a single trusted authority.

Detection of misbehavior. The master's ability to evict deviating (byzantine or rational) players relies on its ability to detect deviations from the protocol. We divide the possible deviations into two categories: *active* and *passive*. An active deviation occurs when a player sends messages that do not coincide with the protocol. By singing all messages with private keys, we achieve non-repudiation, i.e., messages can be linked to their senders and provide evidence of misbehavior, which the master can use to detect deviation.

Passive deviation, which stalls the protocol by withholding messages, is much harder to detect. For example, if the protocol hangs waiting for p_1 to take an action following a message it expects from p_2, we cannot, in general, know if p_2 is the culprit (because it never sent a message to p_1) or p_1 is at fault.

To address this challenge we present our novel DA2A broadcast abstraction, which supports *broadcast*(m) and *deliver(m)* operations for the players and a *detect()* operation for the master. Every player p_i invokes *broadcast*(m) for some message m s.t. all the other players should *deliver*(m). The *detect()* operation performed by the master returns a set S of players that deviate from the protocol together with corresponding proofs:

▶ **Definition 1** (Detectability). *For every two players p_j, p_i s.t. p_i does not deliver a message from p_j, S contains p_j (with a proof of p_j's deviation) in case p_j did not perform broadcast(m) properly, and otherwise, it contains p_i (with a proof of p_i's deviation). Moreover, S contains only deviating players.*

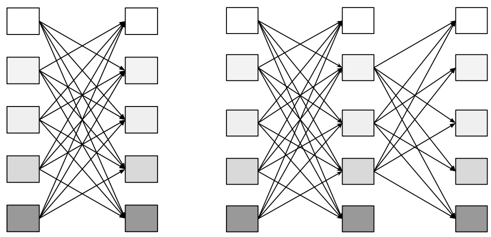
Note that in case S is empty, all the players follow the protocol, meaning that all the players broadcast a message and deliver messages broadcast by all other players.

4 FairLedger Protocol

We present our detectable all-to-all building block in Section 4.1, then use it for our sequencing protocol in Section 4.2, and for the recovery protocol in Section 4.3. In Section 4.4, we informally argue that following the protocol is a Nash equilibrium. For space limitations, the full correctness proof (including game theoretical analysis) is deferred to the full paper [35].

4.1 Detectable all-to-all (DA2A)

Communication patterns. We start by discussing two ways to implement all-to-all communication over reliable links. The simplest way to do so is *direct all-to-all*, in which *broadcast(m)* sends message m to all other players (see Figure 1a). This implementation has the optimal cost of 1 hop and $n(n-1)$ messages, but cannot reveal any information about passive deviations: In case p_i does not deliver a message from p_j, the master has no way of knowing whether p_j did not send a message to p_i, or p_i is lying about not receiving the message.



(a) direct all-to-all **(b)** relayed all-to-all

Figure 1 All-to-all communication patterns.

Another approach, which we call *relayed all-to-all*, designates a subset of the players as *relays*. A *broadcast(m)* sends m to all players, and when a relay receives a message for the first time, it forwards it to all players (see Figure 1b). With r relays, $(r+1)n^2$ messages are sent.

DA2A implementation. DA2A has two modes: normal and alert. Every instance of DA2A starts in the normal mode, in which a broadcast uses direct all-to-all and also informs the master of the broadcast. A *detect*() operation proceeds follows:

- Wait 2Δ time for all players to inform it of their broadcasts.
- In case inform messages are missing from some subset of players $P \subset \Pi$, *detect*() returns P.
- Otherwise, the master waits 2Δ time to make sure that all messages that had been sent have arrived, and then queries all players if they deliver messages from all players.
- If none of the players complains, *detect*() returns $\{\}$.
- Otherwise, the master picks a player p_i that did not deliver a message from player p_j and instructs all players to switch to the alert mode in which they re-broadcast their messages using relayed all-to-all with $2f+1$ players different from p_i and p_j acting as relays.

- After waiting 2Δ time, the master again queries all players if they deliver messages from all players. For every two players p_j and p_i s.t. p_i does not deliver a message from p_j, the master asks the relays whether they received a message from p_j. The relays' replies are signed and used as proof of a deviation. In case $f+1$ relays say yes, the return set includes p_i. Otherwise, it includes p_j.

Correctness. We now prove the detectability property (Definition 1) of our DA2A broadcast.

▶ **Theorem 2.** *If no more than $f+1$ players deviates from the protocol, then (1)* detect() *never returns a player that does not deviate and (2) for every two players p_i, p_j s.t. p_i does not deliver a message from p_j,* detect() *returns either p_i or p_j.*

Proof. Consider two players p_j and p_i s.t. p_i does not deliver a message from p_j in the alert mode. In case $f+1$ relays tell the master that they received a message from p_j, then by the protocol *detect*() includes p_i in its return set, and otherwise it includes p_j. Since p_i does not deliver a message from p_j, we get that either p_i or p_j deviated. Thus, since the master picks $2f+1$ relays other than p_i and p_j, we get that no more than f relays deviate. Therefore, whenever $f+1$ relays report that they received a message from p_j, at least one non-deviating relay forwarded the message from p_j to p_i, meaning that p_i deviated by not delivering it. In addition, since we have $2f+1$ relays, at most f of which deviate, we get at least $f+1$ are not deviating. Therefore, in case fewer than $f+1$ relays report that they received a message from p_j, we get that p_j did not send its message to all relays, i.e., has deviated. ◀

4.2 Sequencing protocol

The sequencing protocol works in *epochs*, where in each epoch every participating player gets an opportunity to append one transaction (or one fixed-size batch of transactions) to the log. To ensure fairness, we commit all the epoch's transactions to the log atomically (all-or-nothing). Recall that we assume that players always have transactions to append.

An *append*(t) operation locally buffers t for inclusion in an ensuing epoch, and waits for it to be sequenced. Each epoch consists of three DA2A communication rounds among players participating in the current epoch (see Figure 2), proceeding as follows:

1. Broadcast a transaction from the local buffer; upon receiving transactions from all, order them by some deterministic rule and sign the hash h of the sequence.
2. Broadcast h; receive from all and verify that all players signed the same hash.
3. Broadcast $\langle commit, epoch, h\rangle$ (signed), and append to local ledger (and return) when receive the same message $f+1$ times.

If any messages are not received, the protocol hangs. The purpose of the first round is to broadcast all the transactions of the epoch. The second round ensures safety; at the end of this round each player validates that all other players signed the same hash of transactions, meaning that only this hash can be committed in the current epoch. The last round ensures recoverability during reconfiguration as we explain in Section 4.3 below. Note that we achieve fairness by waiting for all players; an epoch is committed only if all the players sign the same hash, and since each player signs a hash that contains its own transaction, we get that either all the players' transactions appear in the epoch, or the epoch is not committed.

Read operations. Since all players make progress together, they all have up-to-date local copies of the ledger. A *read*(l) operation simply returns the last l committed transactions in the local ledger. To make sure byzantine players do not lie about committed transactions, a returned batch of transactions st for epoch k is associated with a *proof*, which is either (1) a newConfig message from the master that includes st (more details below), or (2) $f+1$ epoch k round 3 messages, each of which contains a hash of st.

Asynchronous broadcast. The first round of our sequencing protocol exchanges transactions (data), the second round exchanges hashes of the transactions (meta-data), and the last round exchanges commit messages (meta-data). Hence, the first round consumes most of the bandwidth. In order to increase throughput, we decouple data from meta-data and asynchronously broadcast transactions (i.e., execute the first round) of every epoch as soon as possible. However, in order to be able to validate transactions, we perform rounds 2 and 3 sequentially.

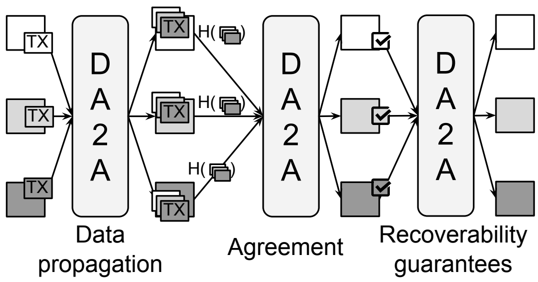


Figure 2 Sequencing protocol.

In other words, we divide our communication into a data path and a meta-data path, where the data path is out-of-order and the meta-data path orders the data. This is a common approach, used, for example, in atomic broadcast algorithms that use reliable broadcast to exchange messages and a consensus engine to order them [13, 20].

4.3 Recovery

To detect deviations that prevent progress, we use the *detect*() operation exposed by DA2A. Recall that the sequencing protocol is an infinite sequence of DA2A instances. Therefore, the master sequentially invokes detect() operations in all DA2A instances. If it returns a non-empty set S, the master invokes reconfiguration.

During reconfiguration the master first stops the current configuration and learns its closing state by sending a reconfig message to the current committee. To prove to the players on the committee that a reconfiguration is indeed necessary, the master attaches to the reconfig message proof reconfiguration is warranted. This can be evidence of active deviation, or a proof of passive deviation returned from DA2A *detect*(). When a player receives a reconfig message, it validates the proof for the reconfiguration, sends its local state (ledger) to the master, and waits for a newConfig message from the master. When a player receives newConfig with a new configuration, it validates that every player removal is justified by a proof, and ignores requests that do not have a valid proof.

State transfer. Note that while a byzantine player cannot make the master believe that an uncommitted epoch has been committed (a committed epoch must be signed by all the epoch's players), it can omit a committed epoch when asked (by the master) about its local state. Such behavior, if not addressed, could potentially lead to a safety violation: suppose that some byzantine player p does not broadcast its last message in the third round in epoch k, but delivers messages from all other players. In this case, p has proof that epoch k is committed, and may return these transactions in response to a read. However, no other player has proof that epoch k is committed and p withholds epoch k's commit from the master. In this case, the new configuration will commit different transactions in epoch k, which will lead to a safety violation when a *read* operation will be performed.

The third round of the epoch is used to overcome this potential problem. If the master observes that some player receives all messages in the second round of epoch k, it concludes that some byzantine player *may* have committed this epoch. Therefore, in this case, the master includes epoch k in the closing state. Since the private keys of byzantine players are unavailable to the master, it signs the epoch with its own private key, and sends it to all players in the new configuration (committee) as the opening state. A player that sees an epoch with the master's signature refers to it as if it is signed by all players. (Recall that the master is a trusted entity, emulated by a BFT protocol.)

4.4 Rationality – proof sketch

We now informally argue that following the protocol is an equilibrium for all rational committee players. The formal proof of appears in the full paper [35].

Since a round 2 message is required from all committee members in order for an epoch to be committed, and since no committee member will sign a hash on a sequence that excludes its transaction (otherwise its ratio in the ledger will decrease), we get that a player on the committee cannot be excluded from a committed epoch. Therefore, players cannot increase their ratio in the ledger by active deviation. Moreover, since the master may punish them for an active deviation by removing them from the committee, following the protocol dominates any active deviation.

As for passive deviations, a possible strategy for a rational player p_i is to try to "frame" another player p_j and get it removed by the master, in which case p_i's ratio in the ledger will grow. It can try to do this by not sending messages to p_j or by lying about not delivering p_j's messages. In order to prove Nash equilibrium we need to show that if all rational players but a player p_i follow the protocol, then even if all f byzantine players help p_i (and so $f + 1$ players deviate from the protocol), p_i still cannot frame another player and get it removed: This follows from Theorem 2.

Moreover, since we assume that among ledgers with the same ratio players prefer longer ones, sending protocol messages as fast as possible dominates slower sending.

5 FairLedger implementations

We implement FairLedger based on Iroha's framework, written in C++. For better comparison we only change Iroha's consensus algorithm (called Sumeragi [46]) with our sequencing protocol, while keeping other components almost untouched (e.g., cryptographic components, communication layer, and client API). This implementation is described in Section 5.1.

In order to evaluate the FairLedger protocol itself, independently of the Hyperledger framework, we implement another version of FairLedger's sequencing protocol based on PBFT's code structure, written in C++ as well, as described in Section 5.2.

5.1 Hyperledger implementation

The Hyperledger framework consists of two types of entities, *players* (committee members in our case) that run the protocol, and *clients* that generate transactions and send them to players for sequencing.

The FairLedger protocol at each player is orchestrated by a single thread, referred to as *logic thread*. The logic thread receives transactions from clients as well as messages from other players into a wait-free incoming event queue. The connections between clients and players are implemented as GRPC sessions [30] (internally using TCP) sending Protobuf messages [29]. The logic thread maintains a map of epoch numbers to epoch states. An epoch state consists of verified events of that epoch, one event slot per player.

Upon receiving a new message, the logic thread verifies it and decides based on the epoch state whether it needs to broadcast a message to other players. Whenever broadcast is required, the logic thread creates and signs the new message, determines the set of its destinations (based on the epoch state), and creates send-message tasks, one per destination. These tasks are handed over to a work-stealing thread pool, in which each thread communicates with its destination over a GRPC connection (See Figure 3).

Iroha is built in a modular fashion, which allows us to swap Sumeragi with FairLedger in a straightforward way. Our evaluation (in Section 6.2) shows that additional Iroha components beyond the consensus engine adversely affect performance. Yet, these components are essential for Hyperledger. For example, Iroha supports multiple operating systems (including Android and iOS) and can be activated from java script code (via a web interface). Such features are essentials for client-facing systems like Iroha, and using standard libraries such as GRPC enables simple and clean development, which is less prone to bugs.

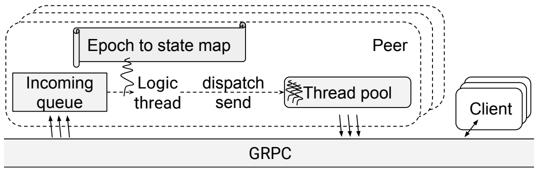


Figure 3 FairLedger implementation in Hyperledger.

5.2 Standalone implementation

To eliminate the effect of the overhead induced the Hyperledger framework, we further evaluate the FairLedger protocol by itself, independently of the additional components. To this end, we employ the PBFT code [17] as our baseline. PBFT uses UDP channels, and is almost entirely self-contained, it depends only on one external library, for cryptography.

In this implementation of FairLedger, the logic thread directly communicates with clients and players over UDP. As in our Hyperledger implementation, the logic thread uses a map of epoch numbers to epoch states, and follows the same logic for generating messages.

Using UDP requires us to handle packet loss. We use a dedicated timer thread that wakes up periodically, (after a delay determined according to the line latency), verifies the progress of the minimal unfinished epoch, and requests missing messages from the minimal epoch if needed.

6 Evaluation

We now evaluate our FairLedger protocol using the two prototypes. The Hyperledger prototype is comparable to Iroha, and the standalone prototype is comparable to PBFT.

6.1 Experiment setup

Configuration. We conduct our experiments on Emulab [48]. We allocate 32 servers: 16 Emulab D710 machines for protocol players, and 16 Emulab PC3000 machines for request-generating threads (clients). Each D710 is a standard machine with a 2.4 GHz 64-bit Quad Core Xeon E5530 Nehalem processor, and 12 GB 1066 MHz DDR2 RAM. Each PC3000 is a single 3GHz processor machine with 2GB of RAM.

Given that our system is intended for deployment over WAN among financial institutions, we configure the network latency among players to 20ms. In Emulab, the communication takes place over a shared 1Gb LAN, denoted S-LAN. Each client is connected to a single (local) player with a zero latency 1Gb LAN. In case clients need to communicate directly with remote players (as they do in Iroha's design), they do so over S-LAN, i.e., with a latency penalty. We benchmark the system at its throughput saturation point.

In our Hyperledger prototype evaluation, we use version v0.75. Since in normal mode we assume no byzantine behavior, we configure Iroha with no faulty players, so it signs each transaction once. The request-generating threads create transactions formatted according to Iroha's specification (given in Protobuf), which consists of a few hundreds of bytes of data.

In our standalone prototype evaluation, we create packets of a similar size, namely 512B of data, as this is the transaction size in our expected use case.

Test scenarios. We compare Iroha and PBFT to FairLedger's two operation modes – the failure-free normal mode and the alert mode activated in case of attacks.

We evaluate the alert mode both under attack of a single byzantine player, and without an attack. In the alert mode we assume that $f=1$, and hence employ 3 relays. In the attack scenario the byzantine player remains undetectable by the master. Specifically, one of the relays withholds messages that it needs to send to one of the other relays.

6.2 Hyperledger

In order to deal with f failures, FairLedger needs $2f+3$ players, and Iroha needs $3f+1$. Therefore, we scale our evaluation from 5 to 9 players. Iroha's clients perform asynchronous operations, and so the operation latency is always zero. Hence, we focus this comparison on throughput.

Figure 4 compares the two modes of FairLedger with Iroha. Results show that FairLedger's normal mode has much higher throughput (up to 3.5x) than Iroha's and the difference grows with the number of players. In both algorithms, due to the usage of GRPC, the bottleneck is the broadcast. FairLedger commits more transactions per broadcast, since each epoch consists of one message from every player, whereas Iroha pays the cost of broadcast for every client request. Therefore, Iroha suffers more as the broadcast cost increases (as we have more players to send messages to).

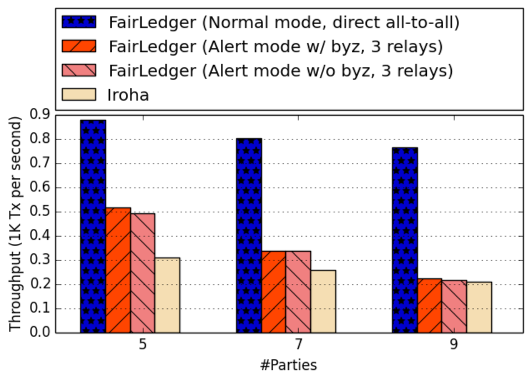


Figure 4 Throughput of FairLedger and Iroha over simulated WAN.

FairLedger's alert modes incur a 44% reduction in throughput with 5 players, and even more as the number of players increases, because the relays worsen the bottleneck by issuing additional broadcast operations. Byzantine behavior slightly improves performance since withholding messages reduces the load on the relays. However, this effect is negligible.

6.3 Standalone prototype

We evaluate our FairLedger prototype that is based on PBFT's code structure. We configure PBFT parameters in a way that maximizes PBFT's throughput, enabling batching and enough outstanding client-requests to saturate the system. We indeed achieve similar results to those reported in recent work running PBFT over WAN [40]. Again, since in order to deal with f failures PBFT requires $3f+1$ players and FairLedger $2f+3$, we run the evaluation with 7 to 16 players. Figure 5 shows the throughput and latency achieved by the protocols.

First, we observe that the absolute throughput is 5x higher than with Iroha. This is thanks to PBFT's optimized bare-metal approach, which sacrifices modularity and maintainability for raw performance. We further see that FairLedger's normal mode has higher throughput than PBFT. This is because PBFT's clients are directed to a single player (referred to as primary or leader), while FairLedger's clients address their nearest player, distributing the load evenly among them.

FairLedger's alert mode with three relays reduces throughput by 30%-40% compared to the normal mode. Note that with 7 players, PBFT achieves about 16% higher throughput than FairLedger's alert mode, but as the number of players increases, the gap closes, reaching 9% lower throughput than PBFT's with 16 players.

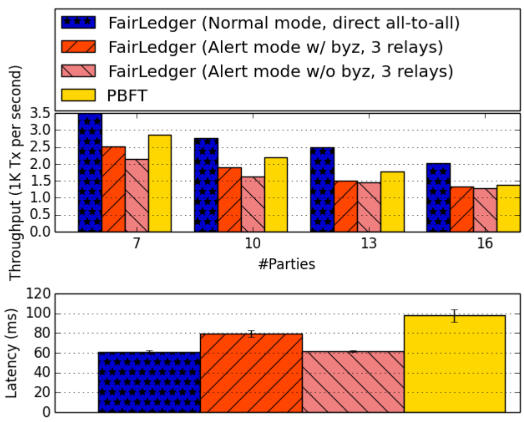


Figure 5 Throughput and latency of FairLedger and PBFT over simulated WAN.

We measure latency below the saturation point. The results for all configuration sizes are similar, and so we depict in Figure 7 only the results with 10 nodes. Error bars depict the standard deviation. The average latency of FairLedger clients in the normal mode is 64ms, which is close to the network latency of 3 rounds of 20ms. Indeed when communicating over WAN, the performance penalty of signing and verifying signatures is negligible. PBFT's average latency is about 106ms, and consists of 3 PBFT rounds and 2 client-primary communication steps.

The average latency of FairLedger's alert mode with a byzantine relay is 86ms, since it consists of 4 rounds of communication. The reason is that one player is always one round behind the rest due to missing the byzantine player's message. Since in the third round he require messages from $f+1$ players (and not all of them), there is no need to wait for the lagging player's round 3 message, and the epoch ends after 4 rounds. The latency of the alert modes without byzantine players is 64ms, similarly to the normal mode.

7 Related Work

Fairness and rationality. Our work is indebted to recent works that combine game theory and distributed systems [2,3,5,9,24,25,36,41,47] to implement different cooperative services. In particular, we adopt a BAR-like model [5,36,41]. As in previous works on BAR fault tolerance [5,36], we assume non-colluding rational players, whereas colluding players are deemed byzantine. As in [41], we do not assume altruistic players – all non-byzantine players are rational in our model.

Practical byzantine fault tolerant consensus protocols [1,6–8,15,16,18,23,32,37–40,49] have been studied for more than two decades, but to the best of our knowledge, only three consider some notion of fairness [7,9,40], and only one of which deals with rational players [9].

One of the important insights in Prime [7] is that the freedom of the leader to propose transactions must be restricted and verified by other participants. To this end, Prime extends PBFT [16] with three additional all-to-all communication rounds at the beginning, in which participants distribute among them self transactions they wish to append to the ledger. The leader proposes in round 4 a batch of transactions that includes all sets of transactions it gets in round 3 from $2f + 1$ participants. Since each transaction proposed by some participant is passed to the leader by at least $2f + 1$ participants, its participant may expect its transaction to be proposed. In case a participant send a request and the leader does not propose it for some time T, the participant votes to replace the leader. As a result, Prime guarantees that during synchronous periods every transaction is committed in a bounded time T.

Similarly to FairLedger, Prime uses batching to commit transactions of different participants atomically together, and uses a PKI to ensure fairness and provide proofs that the batches are valid. However, their fairness guarantee is weaker than ours. Since the first three rounds are asynchronous (i.e., participants do not wait to hear from all, but rather echo messages as soon as they receive them), there is no bound on the ratio of transactions issued by different participants that are committed during T. More importantly, Prime assumes that all non-byzantine participants follow the protocol, and we do not see a simple way to adjust to overcome rational behavior. For example, there is no incentive for participants to echo transactions issued by other participants in the first three rounds; to the contrary – the less they echo, the less transactions from other participants will be proposed by the leader.

Honeybadger [40] is a recent protocol for permissioned blockchians, which is built on top of an optimization of the atomic broadcast algorithm by Cachin et al. [13]. It works under fully asynchronous assumptions and provides probabilistic guarantees. Honeybadger assumes a model with n servers and infinitely many clients. In brief, clients submit transactions to all the servers, and servers agree on their order in epochs. In each epoch, participants pick a batch of transactions (previously submitted to them by clients) and use an efficient variation of Bracha's reliable broadcast [11] to disseminate the batches. Then, participants use a randomized binary consensus algorithm by Ben-Or et al. [10] for every batch to agree whether or not to include it in the epoch.

Similarly to FairLedger, they use epochs to batch transactions proposed by different players, and commit them atomically together. Their (probabilistic) fairness guarantee is stronger than the one in Prime: they bound the number of epochs (and accordingly the number of transactions) that can be committed before any transaction that is successfully submitted to $n - f$ servers. However, if we adapt their protocol to our model where we do not consider clients and require fairness among players, we observe that their guarantee is weaker than ours: Since communication is asynchronous, it may take arbitrarily long for a transaction by player p_i to get (be submitted) to $n - f$ players, and in the meantime, other players may commit an unbounded number of transactions. In addition, their protocol uses building blocks (e.g., Bracha's broadcast [11] and Ben-Or et al. [10] randomized consensus) that are not designed to deal with rational behavior. Moreover, rational players that wish to increase their ratio in the ledger will not include transactions issued by other players in their batches.

The only practical work that deals with rational players we are aware of is Helix [9]. However, in contrast to our work, Helix provide only probabilistic fairness guarantees and relies on a randomness beacon.

Finally, it worth noting that Prime, Honeybadger, and Helix are much more complex than FairLedger. Prime's and Helix's description in [7] and [9], respectively, is spread over more than 6 double column pages, and the reader is referred to their full paper versions for more details. Honeybadger combines several building blocks (e.g., the atomic broadcast by Cachin et al. [13]), each of which is complex by itself.

BFT protocols and assumptions. The vast majority of the practical BFT protocols [6, 8, 23, 32, 37–39, 49], staring with PBFT [16] assume a model with n symmetric servers (participants) that communicate via reliable eventually synchronous channels. Therefore, they can tolerate at most $f < n/3$ byzantine failures [26], and cannot accurately detect participants' passive deviations (withholding a message or lying about not receiving it); intuitively, it is impossible to distinguish whether a player maliciously withholds its message or the message is just slow. Since passively deviating participants cannot be accurately detected, they cannot be punished or removed, and thus byzantine participants can forever degradate performance [18], and rational behavior cannot be disincentivize.

We, in contrast, assume synchronous communication, which together with the use of a PKI allows FairLedger to be simple, tolerate almost any minority of byzantine failures, guarantee fairness, detect passive as well as active deviations, and penalize deviating players. FairLedger uses the synchrony bound only to detect and remove byzantine players that prevent progress, allowing it to be very long (even minutes) without hurting normal case performance. To reduce the cost of using a PKI, FairLedger signs only the hashes of the messages. Moreover, in WAN networks the cost of PKI is reduced due to longer channels delays.

As illustrated by works on Prime [7] and Aardvark [18] most BFT protocols are vulnerable to performance degradation caused by byzantine participants. To remedy this, Aardvark focuses on improving the worst case scenario. We, on the other hand, follow the approach taken in Zyzzyva [32], and optimize the failure-free scenario. We take this approach because byzantine failures are rare in financial settings, and one can expect break-ins to be investigated remedied.

We implement FairLedger inside Iroha [45], which is part of the Hyperledger [28] project. Specifically, we substitute the ledger protocol in Iroha, which was originally based on the BFT protocol in BChain [23], with FairLedger. In brief, their protocol consists of a chain of $3f + 1$ participants, where the first $f + 1$ order transactions. To deal with a passively deviating participant that withholds messages in the chain, they transfer both the sender and the receiver (although only one of them deviates from the protocol) to the back of the chain, where they do not take part in ordering transactions. Similarly to FairLedger, they assume synchrony with coarse time bounds and use it to detect passive deviations. However, in contrast to FairLedger, they do no accurately detect byzantine players and punish correct ones as well. Moreover, since the head of the chain decides on the transaction order, Iroha does not guarantee fairness.

Broadcast primitives. In order to detect passive deviation we define DA2A, a new detectable all-to-all communication abstraction. Even though many practical byzantine broadcasts [12–14, 20, 22, 27, 43] were proposed in the past, DA2A is the first to extend its API with a *detect*() method, which accurately returns all misbehaving players.

8 Discussion

Blockchains are widely regarded as the trading technology of the future; industry leaders in finance, banking, manufacturing, technology, and more are dedicating significant efforts towards advancing this technology. The heart of a blockchain is a distributed shared ledger protocol. In this paper, we developed FairLedger, a novel shared ledger protocol for the blockchain setting. Our protocol features the first byzantine fault-tolerant consensus engine to ensure fairness when all players are rational. It is also simple to understand and implement. We integrated our protocol into Hyperledger, a leading industry blockchain for business framework, and showed that it achieves superior performance to existing protocols therein.

We further compared FairLedger to PBFT in a WAN setting, achieving better results in failure-free scenarios.

References

1. Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. In *Operating Systems Review*, 2005.
2. Ittai Abraham, Lorenzo Alvisi, and Joseph Y Halpern. Distributed computing meets game theory: combining insights from two fields. *Acm Sigact News*, 2011.
3. Ittai Abraham, Danny Dolev, and Joseph Y Halpern. Distributed protocols for leader election: A game-theoretic perspective. In *International Symposium on Distributed Computing*, 2013.
4. Ittai Abraham and Dahlia Malkhi. Bvp: Byzantine vertical paxos, 2016.
5. Amitanand S Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. Bar fault tolerance for cooperative services. In *operating systems review*, 2005.
6. Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Customizable fault tolerance forwide-area replication. In *Reliable Distributed Systems, 2007. SRDS 2007.*, 2007.
7. Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing*, 2011.
8. Yair Amir, Claudiu Danilov, Jonathan Kirsch, John Lane, Danny Dolev, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Scaling byzantine fault-tolerant replication towide area networks. In *Dependable Systems and Networks, 2006. DSN 2006*, 2006.
9. Avi Asayag, Gad Cohen, Ido Grayevsky, Maya Leshkowitz, Ori Rottenstreich, Ronen Tamari, and David Yakira. In *A Fair Consensus Protocol for Transaction Ordering*, pages 55–65, 09 2018. `doi:10.1109/ICNP.2018.00016`.
10. Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience. In *PODC*. ACM, 1994.
11. Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 1987.
12. Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 1985.
13. Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Cryptology*. Springer, 2001.
14. Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In *Reliable Distributed Systems, 2005. SRDS 2005. 24th IEEE Symposium on*. IEEE, 2005.
15. Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, 1993.
16. Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, 1999.
17. Miguel Castro, Barbara Liskov, et al. BFT - Practical Byzantine Fault Tolerance (software). `http://www.pmg.csail.mit.edu/bft/#sw`, 2017. [Online; accessed 16-Apr-2017].
18. Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, 2009.
19. CoreOS. etcd – a highly-available key value store for shared configuration and service discovery. `https://coreos.com/etcd/`, 2017.
20. Flaviu Cristian, Houtan Aghili, Raymond Strong, and Danny Dolev. *Atomic broadcast: From simple message diffusion to Byzantine agreement*. Citeseer, 1986.
21. Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.
22. Vadim Drabkin, Roy Friedman, and Alon Kama. Practical byzantine group communication. In *ICDCS*. IEEE, 2006.
23. Sisi Duan, Hein Meling, Sean Peisert, and Haibin Zhang. Bchain: Byzantine replication with high throughput and embedded reconfiguration. In *OPODIS 2014*. OPODIS 2014, 2014.

24 Joan Feigenbaum, Christos Papadimitriou, and Scott Shenker. Sharing the cost of muliticast transmissions (preliminary version). In *Theory of computing*. ACM, 2000.
25 Michal Feldman, Christos Papadimitriou, John Chuang, and Ion Stoica. Free-riding and whitewashing in peer-to-peer systems. *Journal on Selected Areas in Communications*, 2006.
26 Michael J Fischer, Nancy A Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986.
27 Matthias Fitzi and Martin Hirt. Optimally efficient multi-valued byzantine agreement. In *PODC*. ACM, 2006.
28 The Linux Foundation. Hyperledger. `https://www.hyperledger.org/`.
29 Google. Protocol Buffers - data interchange format. `https://github.com/google/protobuf`.
30 Google. GRPC – Open-source universal RPC framework. `http://www.grpc.io/`, 2017.
31 Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX technical conference*, 2010.
32 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, 2007.
33 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *PODC*. ACM, 2009.
34 Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.
35 Kfir Lev-Ari, Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Fairledger: A fair blockchain protocol for financial institutions. *arXiv preprint arXiv:1906.03819*, 2019.
36 Harry C Li, Allen Clement, Edmund L Wong, Jeff Napper, Indrajit Roy, Lorenzo Alvisi, and Michael Dahlin. Bar gossip. In *Operating systems design and implementation*, 2006.
37 Jinyuan Li and David Maziéres. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *NSDI*, 2007.
38 Shengyun Liu, Christian Cachin, Vivien Quéma, and Marko Vukolic. Xft: practical fault tolerance beyond crashes. *CoRR, abs/1502.05831*, 2015.
39 J-P Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.
40 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *CCS*. ACM, 2016.
41 Thomas Moscibroda, Stefan Schmid, and Rogert Wattenhofer. When selfish meets evil: Byzantine players in a virus inoculation game. In *PODC*. ACM, 2006.
42 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.
43 Michael Reiter. The rampart toolkit for building high-integrity services. *Theory and Practice in Distributed Systems*, 1995.
44 Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 1978.
45 Soramitsu. Iroha - A simple, decentralized ledger. `http://iroha.tech/en/`.
46 Soramitsu. Sumeragi - a Byzantine Fault Tolerant consensus algorithm. `https://github.com/hyperledger/iroha/blob/master/docs/iroha_whitepaper.md`, 2017.
47 Vikram Srinivasan, Pavan Nuggehalli, Carla-Fabiana Chiasserini, and Ramesh R Rao. Cooperation in wireless ad hoc networks. In *INFOCOM*. IEEE, 2003.
48 Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*, 2002.
49 Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. *OSR*, 2003.