The design, architecture and performance of the Tendermint Blockchain Network

Daniel Cason
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland

Enrique Fynn
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland

Nenad Milosevic
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland

Zarko Milosevic
Informal Systems
Toronto, Canada

Ethan Buchman
Informal Systems
Toronto, Canada

Fernando Pedone
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland

***Abstract*—Tendermint is the replication engine at the core of Cosmos, a network of proof-of-stake blockchains. In the lifespan of blockchains, Cosmos and Tendermint are mature technologies, currently used by more than a hundred businesses and deployed by hundreds of nodes. The system was designed to provide flexible deployment despite heterogeneous environments, scale performance with the number of nodes, and tolerate misbehaving participants. In this practical experience report, we overview Tendermint's main design goals and architecture, and present a detailed performance evaluation of the system in a realistic environment. We report results from a geographically distributed environment with up to 128 nodes, including failure-free executions and fail-prone scenarios, with both crash and byzantine failures.**

***Index Terms*—Blockchain, Distributed consensus, Byzantine fault-tolerance, Performance and dependability evaluation**

I. Introduction

Tendermint is a state machine replication (SMR) [1] engine that tolerates Byzantine faults. It was among the first systems to adapt classical Byzantine Fault Tolerant (BFT) consensus protocols to the blockchain paradigm, whereby consensus is performed on cryptographic hash-linked batches of transactions (i.e., *blocks*) in a public, open-membership network [2]. Tendermint functions as a blockchain middleware that supports the replication of arbitrary applications, written in any programming language. Tendermint forms a core component of Cosmos [3], a network of independent proof-of-stake blockchains. To date, there are numerous public cryptocurrency networks in production using Tendermint, and more than 200 projects using Cosmos and Tendermint.[1]

Prior to Tendermint, most blockchain systems used the so-called Nakamoto Consensus [4], where blocks are produced by a probabilistic process parameterized by an economic cost. In its original form, cost was measured as the *proof-of-work* (i.e., partial hash collisions) performed by consensus participants known as miners. Due to the high energy cost of proof-of-work [5], a variant known as *proof-of-stake* emerged, where economic cost took the form of stake in the network (i.e., account balances), held by consensus participants known as *validators*. Early concerns about the safety properties of proof-of-stake designs [6] were resolved by the introduction of validator deposits that could be destroyed, or *slashed*, upon evidence of Byzantine behavior [7].

Tendermint is the first system to implement the slashing style of proof-of-stake. Rather than using Nakamoto Consensus, Tendermint's consensus algorithm [8] is a variant of PBFT [9] and of DLS for Byzantine faults with authentication [10], built on top of an efficient gossip layer. Since Tendermint supports open-membership peer-to-peer networking, where it is unlikely that every pair of nodes can communicate directly and nodes may join and leave as they please, gossip [11] plays an essential role in propagating messages through the network. While it was designed for use in public cryptocurrency settings, with validators determined by economic stake, Tendermint can also be used in more private or permissioned settings as a general purpose, production-grade software for BFT state machine replication, where validators are determined according to the application requirements.

In this practical experience report, we overview Tendermint's main design goals and architecture, and present a detailed performance evaluation of the system in a realistic environment, considering failure-free scenarios and scenarios subject to crash failures and Byzantine failures. We highlight below the main conclusions from our performance evaluation.

- Tendermint builds a well-connected communication overlay, used by the gossip layer. While conservative, this setup withstood all faulty scenarios we considered. Moreover, the network swiftly recovered from crashed nodes.
- Tendermint's performance degraded smoothly as the system size increased. In particular, an $8\times$ increase in the system size led to 18% reduction in throughput, better than previously reported on the scalability of SMR systems [12].
- The crash of one third of validators turned out quite harmful to performance, although we did not observe any violations of safety or liveness.
- Inducing Tendermint to misbehave under equivoca-

[1] https://airtable.com/shrapBPVHSLsfhKzk/tblhXmOhbIClTJfL9/viw7FWJKmkIjnaM7w

tion [13], [14], where Byzantine nodes can send conflicting messages to different processes, with an increasing number of Byzantine validators proved difficult.
- This last observation suggests that the overlay nature of peer-to-peer communication provides additional resilience, when compared to a fully connected network.

The remainder of the paper is structured as follows. Section II discusses the system assumptions. Section III presents Tendermint's design goals and summarizes its architecture. Sections IV to VI overview Tendermint's main components. Section VII presents the performance evaluation. Section VIII surveys related work, and Section IX concludes the paper.

II. System Model

Tendermint is a distributed message-passing system composed of a dynamic set of processes or *nodes*. The set of nodes can vary over time as nodes join and leave the network. Nodes interact by exchanging messages via encrypted point-to-point communication channels. A node is not assumed to know the entire set of nodes in the system. A node communicates directly with only a restricted subset of nodes, which are called the node's neighbors or *peers*. To send a message to nodes that are not its peers, a node relies on other nodes to relay the message to its destinations. In this case, communication takes place via gossip [11], [15].

A node is expected to maintain a long-term persistent identity in the form of a public key, from which the node's unique ID is derived. When attempting to connect to a peer, a node verifies whether the peer is in possession of the private key corresponding to its ID, thus preventing man-in-the-middle attacks. A node listens on one or multiple network addresses for connections from peers. The number of connections that a node accepts is typically bound, and some nodes may not accept connections at all. A node also establishes connections with a number of peers, to ensure a minimal connectivity with the rest of the network. By requesting and accepting connections from peers, nodes construct an overlay network that eventually, with high probability, is connected.

Tendermint considers the Byzantine failure model. It tolerates both benign faults, where nodes crash and may later recover, and Byzantine faults, including arbitrary and potentially malicious behavior. A subset of nodes play the role of *validadors*. The set of validators is dynamic and known by all the nodes. Validators execute Tendermint's consensus algorithm [8]. Tendermint assumes that at most one third of the validators are Byzantine. For progress, Tendermint relies on partial synchrony [10]: the system is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous, the Global Stabilization Time (GST), is unknown to the nodes.

III. Design and architecture

Tendermint was designed to support the development of open, public, and general-purpose blockchain applications, deployed in large-scale, geographically distributed, and hostile environments. Tendermint's design was motivated by the following goals.

a) Deployment flexibility: Tendermint should support deployment of blockchain applications with nodes distributed among multiple administrative domains. Administrators should be allowed to decide whether to expose nodes to the public network. As a consequence, Tendermint should not enforce a network setup, not assuming, in particular, that every pair of nodes can communicate directly. The adoption of gossip as communication means, and the design of the peer-to-peer layer (Section IV) derives from this assumption.

b) Scalability: Tendermint should enable applications to be replicated among hundreds (possibly thousands) of nodes. Tendermint adopts strategies to avoid substantial performance degradation with increasing number of participants, in spite of the performance degradation of consensus-based state machine replication at scale [12], [16], [17].

c) Byzantine fault tolerance: Tendermint is expected to operate in hostile environments. In addition to a consensus protocol that tolerates Byzantine validators (Section V-B), all auxiliary protocols should also consider malicious behavior. Moreover, a specific protocol (Section V-C) was implemented for nodes to exchange evidences of misbehaviour, which may lead Byzantine validators to have their deposits slashed.

d) Language independence: Tendermint should support the replication of applications written in any programming language. For this, the interaction of the Tendermint with the replicated application should be via a standardized, language-agnostic protocol, using the Application BlockChain Interface (ABCI), briefly summarized in Section VI.

e) Light clients: Tendermint must support resource-constrained nodes (light clients [18]) that cannot download the entire blockchain. Yet, light clients must be able to verify hashes, signatures, and membership changes, detect and react to Byzantine behavior, among other activities. Due to space constraints, we do not consider light clients in this paper.

Figure 1 presents Tendermint's architecture. A peer-to-peer substrate provides communication for the main blockchain modules. A client interacts with the system via RPC by submitting transactions that are added to the mempool module, and receiving a response generated by the consensus module. The consensus module is in charge of ordering and intermediating the execution of transactions, by means of the application interface. In the following sections, we comment on each one of these modules.

IV. Peer-to-Peer Communication

A Tendermint node is not expected to establish direct connections to all other nodes in the network, but only to a subset of them, called its peers. It is up to the peer-to-peer communication layer to maintain an overlay network that allows all nodes to exchange messages through gossip. More specifically, the peer-to-peer communication layer has the following responsibilities: (a) the peer discovery service; (b)

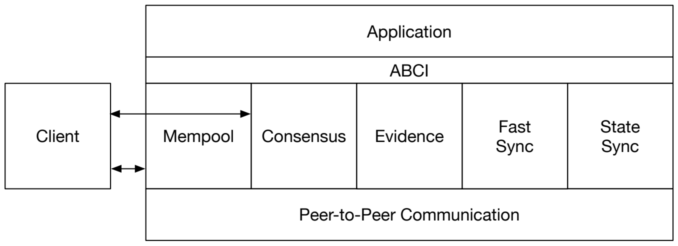

Fig. 1. Tendermint's architecture.

the management of an address book of peers and peer's quality statuses; and (c) connection establishment and management.

Tendermint provides several options for peer discovery: (i) using a bootstrap service, implemented by *seed nodes*, who continuously explore the network in order to maintain a comprehensive list of active nodes, and provide random sample to nodes upon request; (ii) a *persistent peers* mechanism, where a node to join the Tendermint network is configured with the IDs and network addresses of a number of peers with which it tries to maintain connections; and (iii) the Peer Exchange (PEX) protocol, by which nodes periodically exchange lists of potential peers with nodes they are connected to.

The set of peers a node knows about is stored in the node's *address book*, which maintains peer ID, associated network addresses, and parameters representing the quality of a peer. The node's address book is fed with potential peers by the peer discovery service, and records all peers with which it has established connections. A node can define quality metrics of its peers based on its behavior in the multiple Tendermint protocols (Section V), where receiving relevant messages from the consensus protocol is the most important criteria. A peer can be removed from the node's address book and blacklisted when it does not follow the protocol, for instance, by sending unsolicited messages, or sending messages in a higher rate or in shorter intervals than the configured ones.

A node in Tendermint establishes persistent, encrypted connections with its peers. The connection manager routine has the goal of maintaining a target number of *outbound* connections with peers (10 by default), which includes the configured persistent peers. While this target is not reached, or when outbound connections are dropped for any reason, the node retrieves potential peers from the address book and tries to establish connections with them. In addition, a node accepts a maximum number of *inbound* connections (40 by default) from peers. Once this limit is reached, further connection requests are rejected. To enable nodes to exchange messages from multiple protocols using the same connection, Tendermint defines the abstraction of *channels*. Channels have configured priorities, and maximum sending and receiving rates. In short, the priority of a channel defines which messages will be sent first through a connection, when there are pending messages from multiple channels.

V. Tendermint Core

Tendermint comprises a number of modules (see Figure 1), and a node may participate in all or in only some of them. This section describes the main modules.

A. Mempool

The *mempool*, or transactions pool protocol is an entry point of Tendermint. It receives, validates, stores, and broadcasts transactions submitted by clients. These transactions, provided that they are considered valid by the application, are eventually included in blocks committed by the consensus protocol.

Nodes expose an interface to receive transactions submitted by clients, via RPC calls. Transactions received by a node are sent to the application to be validated. This is done through the *mempool* ABCI connection, presented in Section VI. Once a transaction is validated, the node appends the transaction to the mempool, which is actually a list of valid transactions. For each peer of a node that also participates in the mempool protocol, the node maintains a send routine. Each send routine iterates through the list of transactions and sends them to the peer, in the same order in which they were validated.

Transactions received from peers go through the same process as transactions received from clients: if they are validated by the application, they are appended to the mempool. The only difference is that received transactions are first checked against the cache of recent transactions, to prevent a transaction from being validated multiple times by the application. The cache of recent transactions has a fixed size and only stores transaction hashes, associated to a map of peers from which a transaction was received. This cache is also accessed by send routines, to prevent sending a transaction to a peer from which the same transaction was recently received.

Transactions submitted by clients are therefore broadcast to all nodes, and are eventually received by the validator node that is responsible for proposing the next block for consensus. This node, or more specifically its consensus module, retrieves a list of pending transactions from the mempool to build the proposed block. This functionality is exposed via a method call that returns the longest prefix of the transactions list that respects the limits established by the caller, typically maximum number, size in bytes, and required gas (fee).

When a block is committed by the consensus protocol and all transactions included in the block are executed, the mempool is contacted to update the list of pending transactions. More specifically, the committed transactions are then removed from the mempool, since they are not pending anymore. Their hashes, however, are not removed from the cache of recent transactions, as they can be still be received from peers and should not be added again to the mempool.

The commit of a block updates the state of the application, as a result of the execution of the transactions it included. To prevent inconsistent behavior when validating new transactions, the mempool is locked while transactions are executed. Moreover, after removing the committed transactions from the mempool, the remaining transactions are sent to the application for a new validation. The rationale behind this procedure is that a transaction previously considered valid might become invalid when considering the updated application state.

B. Consensus

The consensus protocol is responsible for deciding the next block of transactions to be appended to the blockchain. Tendermint's consensus algorithm is described and proved correct in [8], [19], [20]. In this section, we present system aspects not discussed in the algorithm's original description.

a) Consensus protocol: Tendermint's consensus protocol runs a sequence of instances of consensus, where each instance decides on a single block of transactions. Each instance or *height* of consensus is typically composed of one or more *rounds* of consensus. Each round of consensus is led by a validator, the round's *proposer*, and is composed of three round steps: *propose*, *prevote*, and *precommit*. The execution of a round of consensus in Tendermint is similar to the failure-free execution of PBFT [9], also composed of three communication steps.[2]

A round starts with the *propose* step, where the round's proposer signs and broadcasts a block of transactions. Upon receiving a block from the proposer of the current round, and provided that the block can be accepted, a validator signs and broadcasts a *prevote* message for the proposed block id (*prevote* step). The acceptance of a block is defined by a set of consensus-specific rules [8], and by an application-specific predicate that indicates whether a block is valid [21], [22]. Upon receiving *prevote* messages for the proposed block id from a *quorum* of validators, a validator signs and broadcasts a *precommit* message for the proposed block id (*precommit* step). Upon receiving *precommit* messages for the same block id from a *quorum* of validators, a validator commits the block, and appends it to the blockchain. The transactions included in the committed block are then delivered to the application, and the validator proceeds to the next *height* of consensus.

A distinctive aspect of Tendermint's consensus protocol is that the role of proposer rotates. So, while in PBFT [9] the proposer is only replaced when it is suspected to be faulty, in Tendermint all validators take turns as proposers in the first round of successive heights of consensus—when they are free to propose any block. If a round of consensus does not succeed, due to asynchrony or failures, a new round led by a different validator is started. Starting new rounds in Tendermint's consensus does not require view-change phases, like in PBFT and other BFT consensus algorithms. Instead, a set of locking/unlocking rules define whether a proposer must re-propose a block accepted in a previous round, and whether a validator should accept the block proposed in a round [8].

Tendermint's consensus protocol allows validators to have distinct *voting powers*, a form of weighted voting consensus protocol [23]. This means that the vote of a validator, both in *prevote* and *precommit* steps of the protocol, may count more than the vote of another validator. This is in line with the *proof-of-stake* concept adopted by Tendermint, where the more stake (funds) a validator binds to the network, the higher its voting power is. A *quorum* is then defined as a subset of validators whose aggregated voting power is larger than $2/3$ of the voting power of all validators. If the voting power is equally distributed, a *quorum* consists of any subset with more than $2/3$ of the validators—like in other BFT consensus algorithms. The voting power also influences the function that assigns proposers to heights and rounds of consensus. While all validators will be selected as the proposer, the frequency with which a validator is assigned as the initial proposer of a height of consensus is proportional to its voting power.

[2]In PBFT those steps are called *pre-prepare*, *prepare*, and *commit*.

b) Validators set: Each height of consensus is run by a given set of validators, which is known by all nodes. A validator is a Tendermint node that holds a cryptographic key pair. The validator's private key is used to sign the consensus messages it broadcasts, while the validator's public key allows to verify the authenticity of its messages. The *validators set* of a height of consensus is then the set of public keys of all active validators, plus their associated voting powers.

The initial validators set is retrieved from the *genesis* state, shared by all nodes in a Tendermint blockchain. This set can be updated through special commands produced by the application when processing a block of transactions. The new, updated validators set is included in the next block proposed for consensus and, once accepted by a *quorum* of validators and committed, is adopted from the next height of consensus (the proposer-selection function is updated as well). In other words, the validators set is dynamic, and all changes in the validators set are stored in the blockchain.

c) Gossip: The communication substrate supporting the consensus protocol does not relay messages in a best-effort manner, like in *mempool* and in other Tendermint modules. Instead, messages are selectively forwarded to peers based on the information that a node has about the state of each peer in the consensus protocol. A node keeps track of the height, round, and round step at which each peer is, from which it infers which messages a peer potentially needs at any given moment. For instance, if a peer is known to be at the *precommit* step of a round of consensus, there is no point in sending to the peer messages from the *prevote* step of the round that it has already concluded. The node will rather send to the peer messages from the *precommit* step of the round that it assumes the peer has not yet received.

The same applies to a node with peers that are not in the same height and round of consensus. If a peer is behind, it should be provided with messages that allow it to reach the current height and round of consensus. After which the node can relay to the peer the latest consensus messages. At the same time, if a peer is ahead in the consensus protocol, there is no point in relaying messages that the peer will discard. In other words, the consensus gossip substrate tries to align nodes with their peers in the consensus protocol, so that they are in the same height, round, and round step of consensus.

C. Evidence

A crucial element of proof-of-stake blockchain networks is the ability to punish misbehaving (Byzantine) validators. This feature is implemented in Tendermint by the *evidence* module, which is responsible for collecting, verifying, and propagating evidence of a node's misbehavior. There are essentially two scenarios that represent a misbehavior for a Tendermint node: (i) publishing to clients blocks that were not committed by the consensus protocol, and (ii) when acting as validator, casting votes for two distinct blocks in the same round of consensus.

When light clients detect scenario (i), as discussed in [18], or (ii) when a Tendermint node detects that a validator voted for distinct values in a round of consensus, they report the misbehavior to the evidence module. The report must include evidence that would prove the misbehavior. We focus here on scenario (ii), where two conflicting votes in the same round of consensus are the evidence of misbehavior, signed by the same validator, for distinct blocks. This is detectable, in particular, because votes are disseminated via gossip.

When the consensus protocol receives conflicting votes from the same validator in a round, the node reports the two votes to the evidence module. The two distinct votes, signed by the same validator, are evidence enough to prove that the validator is misbehaving. In other cases, like in scenario (i), more information must be attached to the report of evidence of misbehavior. Upon receiving a report, the evidence module verifies whether the attached proofs are valid, according to the node's local state. If so, the node adds the evidence, together with its proofs, to an *evidence pool*. The contents of this pool are then gossiped to the Tendermint network.

When a node receives a misbehavior report from a peer, it first verifies the attached proofs against its local state. If it is deemed a valid evidence of misbehavior, it is added to the local evidence pool, whose content is forwarded to the node's peers. As it occurs with submitted transactions, evidence from the evidence pool are included in blocks proposed for consensus. When a block is committed, evidence of misbehavior included in the block are delivered to the application. It is up to the application to apply the necessary measures against misbehaving nodes, in particular when they are validator nodes. The typical approach, adopted in Cosmos in-production blockchains, is to slash the fundings (stake) bound by that validator, while removing it from the validators set.

D. Fast Sync

The Fast Sync protocol allows nodes to exchange committed blocks. It is used by nodes (re)joining the network to catch up with its peers, by downloading committed blocks from them, ideally in parallel. This is an alternative to learning the blocks decided in every instance of consensus, from the last height on which the node participated (if any) to the current one.

Fast Sync is a peer-to-peer protocol that distinguishes the roles of client and server. Clients are nodes that recently joined the network, or recovered from a period of inactivity. Servers are nodes that have blocks requested by clients, i.e., blocks already learned, executed, and committed.

A client node in the Fast Sync protocol periodically broadcasts *StatusRequest* messages to all peers. A node, regardless of its role, responds with a *StatusResponse* message containing the range of contiguous available blocks. The upper limit of this range is the node's current height, the last block it has committed and executed. Based on this information, a client node selects peers to which it sends *BlockRequest* messages, referring to a specific height. A node responds to it with the requested block, in a *BlockResponse* message, or with a *NoBlockResponse* message, when the requested block is not available. It is worth nothing that all these messages are sent in a best-effort manner, to avoid flooding the peer.

Blocks retrieved via Fast Sync are stored in a buffer, as they are not necessarily received in order. They are executed and committed in order, however, following the same procedure adopted for blocks decided by the consensus protocol. When a client catches up with their peers (i.e., it has received and applied all blocks its peers have already committed), the procedure is concluded and the node switches to the regular operation, using the consensus protocol.

Fast Sync is sensitive to attacks from malicious nodes, since it transfers a considerable amount of data between peers. At the same time, it is useful as long as several peers are willing to collaborate in the state transfer. The protocol tries to mitigate the damage from attacks by limiting the send rate on channels, and by assuming that correct nodes equally distribute requests among peers. However, besides disconnecting from peers that take too long to respond, no protection mechanism is devised.

E. State Sync

The State Sync protocol allows nodes to rapidly bootstrap and join the network in a relatively updated state, by discovering, fetching, and restoring application-level snapshots. The protocol is an alternative to joining consensus at the *genesis* state and replaying all historical blocks, either via consensus or via Fast Sync, until the node catches up with its peers. By relying on State Sync, however, a node will have a truncated history, lacking the ability to audit the blockchain.

State Sync is also a peer-to-peer protocol with clear roles of client and server. A client is a node joining the network and looking for an application snapshot, taken on the greatest possible height. A server is any node in regular operation, whose associated application instance has taken snapshots. The protocol is initiated by a node in client mode that sends snapshot request messages to each new peer with which it establishes a connection—since the client node is starting up, it is trying to connect to an initial set of peers.

Upon receiving a snapshot request message, a node in server mode contacts the application, namely the instance of the replicated application it hosts, to retrieve a list of available snapshots. The interaction of a node with the application is done via the *snapshot* ABCI connection, presented in Section VI. The application should return a list of snapshot descriptors that are sent back to the peer. Upon receiving a snapshot response from a peer, a node in client mode adds the reported snapshot descriptor to a pool of candidate snapshots.

This pool is periodically consulted by the node, which selects the best candidate, essentially the one from the greatest height.

Once a client node has a candidate snapshot, it sends its descriptor to the application, which may accept or reject the snapshot. In particular, the snapshot root hash should match the application hash stored in the block at the height in which the snapshot was taken. If it is accepted, the node requests the multiple snapshot chunks in parallel, possibly from different peers. Note that multiple peers may have offered the same snapshot, i.e., reported the same snapshot descriptor, potentially improving the snapshot retrieval speed. The node requests a snapshot chunk from a peer by sending it a chunk request message, specifying the snapshot height and chunk index. Upon receiving a chunk request message, a node in server mode retrieves the requested snapshot chunk from the application, and provides it in a chunk response message.

Snapshot chunks are not necessarily received in order by a client, but they are offered to the application following the order established by their indexes. Once a full snapshot is retrieved and installed by the application, the node will be at height of the blockchain when the snapshot was taken, and can switch to the Fast Sync or to the consensus protocol.

VI. Application interface

ABCI is the interface between Tendermint and the replicated application. A node maintains four ABCI connections with the replicated application. The *consensus* connection is used for the execution of blocks, and also allows the application to submit commands to reconfigure the set of validators (Section V-B) and to configure other consensus-related node parameters. The *mempool* connection is used by the transaction pool protocol (Section V-A) to validate transactions submitted by clients against the application state. It is up to the application to define whether a transaction is valid or not, and the validation is optional. The *snapshot* connection is used by the State Sync protocol (Section V-E) to serve and restore snapshots of the application state. The application is expected to periodically take snapshots of its state and to persist them to disk. The *query* connection allows retrieving information from the local instance of the application, used by several Tendermint modules (e.g., peer filtering).

VII. Evaluation

We conducted experiments in a geographically distributed environment, with validator nodes evenly spread among 16 AWS regions in all continents. An additional AWS instance hosted a non-validator node, operating in *seed* mode, and all clients. This additional node had two roles: First, it provided to the validators lists of potential peers to which they can establish connections, forming the network. Second, it delivered to clients the sequence of blocks committed by Tendermint, so that clients can compute latencies and throughput. We colocate all clients in a single AWS server to centralize all measurements in a single location. Clients submitted 1KB transactions to the validators evenly, in closed-loop. This means that a client submits a transaction to a given validator, waits until the transaction was included in a block committed by Tendermint, and then submit a new transaction to the same validator.

We used Tendermint version 0.33.8 and Go version 1.15, with default configurations. The mempool can store up to 5000 transactions, with maximum byte size of 1GB, and the block size is 20MB. These default limits are enough to cope with the maximum number of outstanding transactions. Both connection's maximum send and receive rates are 5000 KB/s, and the intervals that govern gossip are 100ms.

A. Network setup

We evaluated the characteristics of the network generated by Tendermint when relying on a *seed* node. Figure 2 depicts the distributions of two metrics of the generated overlay network with 128 nodes. On the left, the number of neighbors a node has in the overlay network, that is, the number of connections it establishes with peers. On the right, the distance between two nodes in the overlay network, that is, the minimal number of communication steps (hops) between them. The cumulative distributions are represented by the green lines, while the bars represents the portion of values in each x-axis interval.

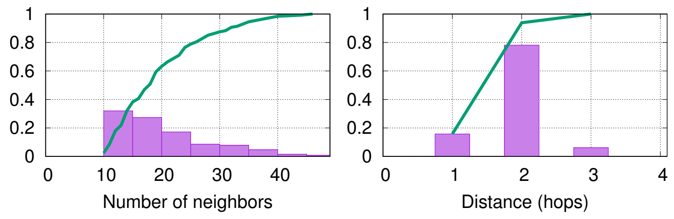

Fig. 2. Distributions of the number of neighbors per node (left) and the distance between two nodes (right) in a Tendermint network with 128 nodes.

Figure 3 presents the distribution of measured latencies between the 16 AWS regions hosting validator nodes (left) and the approximate distribution of latencies between nodes (right), computed as the shortest-path between each pair of nodes in the overlay network and the inter-region latencies. The two distributions would match if we had a fully connected overlay. Tendermint's overlay introduces delays in the communication (e.g., while $\sim$76% of the regions have a delay below 100ms, only 40% of the nodes are below 100ms).

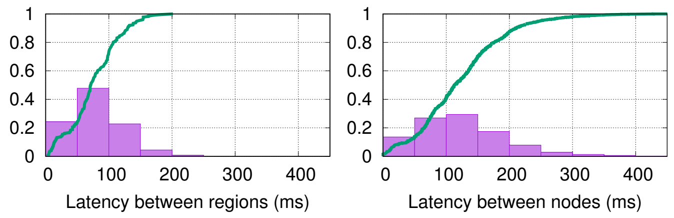

Fig. 3. Distributions of measured latencies between the 16 regions (left), and of expected latencies between the 128 nodes in the overlay network (right).

B. Baseline performance

We evaluated the performance of Tendermint in a network with 128 validator nodes. In the experiments, we start the clients after the network is generated and stable. Clients are evenly distributed among the validators, to which they submit 1KB transactions until 30 blocks of transactions are committed. We subjected Tendermint to increasing workloads by varying the number of clients. We found that a workload with 1536 clients, 12 per validator, provides the best balance between throughput and average latency, while saturating the system. It is hereafter referred to as *reference workload*.

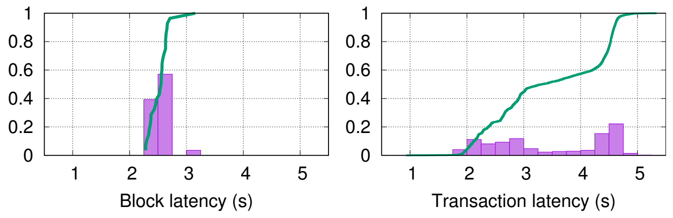

Fig. 4. Distributions of latencies of committed blocks (left) and of submitted transactions (right) with 128 validators.

Figure 4 depicts the distribution of latencies for Tendermint with 128 validators under the reference workload. On the left, the time required for committing a block of transactions, that is, the difference of timestamps of successive blocks. On the right, the time it takes for a client transaction to be ordered, from its submission to the delivery of the block that includes it. New blocks are committed at regular intervals: it takes a block on average 2.53s to commit, with 96% of them between 2.3s and 2.7s. The latencies of submitted transactions present more variability, 3.45 ± 0.99s on average, concentrated in two intervals. About 40% of the latencies were between 2s and 3s, in line with the typical block latency, while a similar portion were around 4.5s, which is closer to the average duration of two blocks. This indicates that while some transactions are included in the first block following their submission, other transactions are shifted to the subsequent block.

It is worth noting that block latencies include an artificial delay of 1s, the *timeout commit*, intended to collect as many votes as possible endorsing the committed block. The goal is to record all validators that contributed to the commit of each block, which might then be rewarded by the application. The remainder of the block latencies (around 1.5s) encompass the three communication steps of the consensus algorithm (see Section V-B).

Figure 5 presents the throughput achieved by Tendermint with 128 validators under the reference workload. On the left, we present the instantaneous throughput when every block is delivered (line with points) and every two blocks (green line). The former presents a lot of variation between blocks, which is attenuated in the second representation, around the overall throughput of 438 transactions per second (tps). On the right of Figure 5, we present the distribution of the number of transactions included in each block. As block latencies are

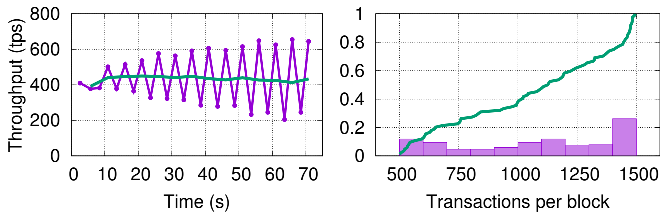

Fig. 5. Instant throughput every block and every two blocks (left), and distribution of the number of transactions committed in each block (right).

similar, the zigzag variation in the instantaneous throughput is due to the number of transactions committed in each block.

C. Scalability

We compare the performance of Tendermint at scale, with 16, 32, 64, and 128 validators. We do not expect performance improvements by increasing the number of validators, since the message complexity and the cost of consensus increase with the number of processes. The reference workload for 128 validators was used in this comparison, by evenly distributing 1536 clients among the validators in the experiment. The same experimental setup considered in Section VII-B was adopted.

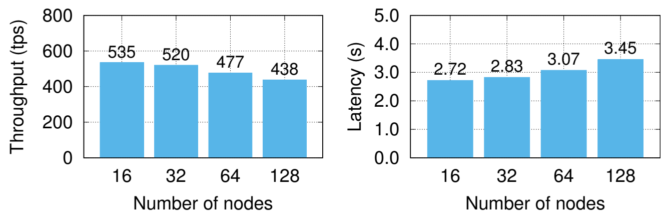

Fig. 6. Performance summary when varying the number of validators under the same workload: overall throughput (left) and average latency (right).

Figure 6 compares overall throughput and average latency of Tendermint as we vary the number of validators. Under the same workload, throughput degrades gracefully as we double the number of validators. From $N = 16$ to $N = 32$, 64, and 128, throughput drops, respectively, by 3%, 11%, and 18%. From both $N = 32$ to 64 and $N = 64$ to 128, throughput drops by 8%. As the number of transactions submitted is the same for all system sizes, the throughput degradation is directly linked to increased block latencies. In fact, while the average block latency with $N = 16$ was 2.14s, it increased to 2.20s (+3%), 2.38s (+11%), and 2.53s (+18%) with $N = 32$, 64, and 128.

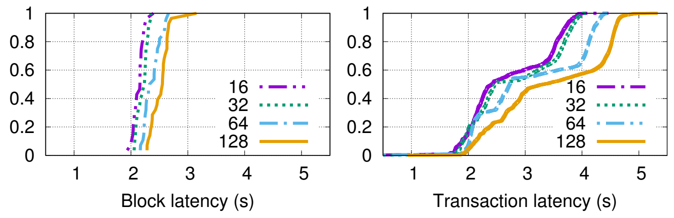

Fig. 7. Distributions of latencies of committed blocks (left) and of submitted transactions (right), when varying the number of validators.

Figure 7 compares the distributions of block latencies (left) and transaction latencies (right) as we doubled the number of validators. The already mentioned increase in block latencies is somewhat expected, as the message complexity of consensus is quadratic on the number of participants. This, however, does not fully explain the degradation of transaction latencies, that from $N = 16$ to $N = 32$, 64, and 128 had averages increased by, respectively, 4%, 13%, and 27%. As observed in Figure 7 (right) for all system sizes, a portion of the transactions were not received in time to be included in the next block, a behavior discussed in Section VII-B. As we increased N, more transactions presented this behavior and, in general, more variable were the latencies. We attribute this to increased delays to propagate transactions to all validators via the *mempool* protocol with increasing network sizes.

D. Crash faults

We evaluated Tendermint under crash faults, considering the setup used in Section VII-B, with 128 validators and 1536 clients, but doubling the experiment duration, from 30 to 60 blocks. After committing 12 blocks, at instant 33s of the experiment, we killed 42 validators. This is the maximum number of crashed validators that would still allow the system to progress. Clients assigned to crashed validators are, after a timeout, re-routed to another validator chosen at random. We first analyze the impact of crashed validators on the network, in terms of connectivity, then on the overall performance.

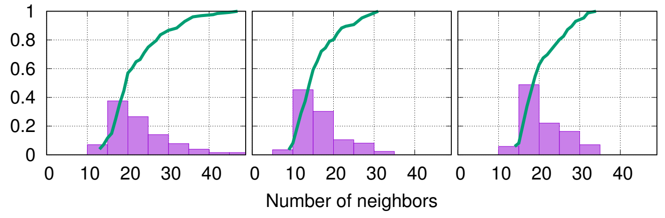

Fig. 8. Evolution of distributions of the number of neighbors per node: before crashes (left), after crashes (center), and when the network re-healed (right).

Figure 8 presents the evolution in the number of neighbors per node at system startup (left), after 42 nodes have crashed (center), and when connections were reestablished (right). A first observation is that the Tendermint network, due its high connectivity, remains connected with the removal of almost 1/3 of the nodes. We considered several methods to select the nodes to crash, and adopted the one with the highest impact on connectivity. From the 1423 connections at startup (22.2 per node), 688 connections remained after the crashes (16.0 per node, with 86 nodes), a 52% reduction. It took around 6s for nodes to detect the failure of all peers, when this second distribution, on the center of Figure 8, was built. Meanwhile, nodes that lost connections tried to establish new ones: 199 new connections were established within 137s, increasing the average number of neighbors per node to 20.7, as depicted on the right of Figure 8. This shows the ability of the PEX protocol to reestablish the network connectivity.

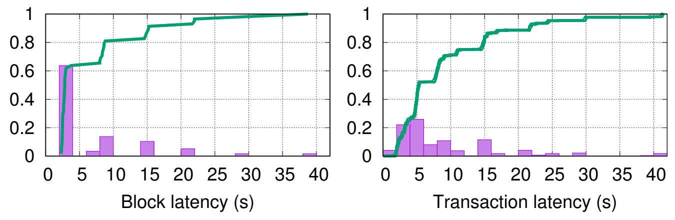

Fig. 9. Distributions of latencies of committed blocks (left) and submitted transactions (right) with 128 validators, from which 42 nodes have crashed.

Figure 9 presents the distributions of block latencies (left) and transaction latencies (right) in the crash-fault experiment. The main impact of having almost 1/3 of the validators crashed is that latencies are not concentrated in a short interval, as in Figure 4, but grouped into six time ranges. The first range, accounting for about 64% of blocks, contains latencies similar to the ones observed in the fault-free scenario, around 2.65s. This means that the consensus instances that decided those blocks were not affected by the crashes. To understand the behavior of the other blocks, we recall that in the Tendermint consensus algorithm [8], the *proposer* role rotates. That is, each instance (height) or round of consensus is assigned to a different validator. If the proposer assigned to the first round of an instance is crashed, an additional round of consensus is required, being assigned to another validator. If it also fails to propose a block, a further round is required, and so on.

The consequence of this procedure is observed in Figure 9: about 17% of the blocks required 2 rounds to be committed, with latencies around 8.4s; 10% required 3 rounds, latencies around 15s; 5% required 4 rounds, latencies around 22s; and 4% required 5 and 6 rounds, latencies around 30s and 39s. Notice that the duration of unsuccessful rounds increases: second rounds had duration around 5.8s, third rounds around 6.6s, and so on. This happens because both faulty proposers and failed rounds are detected via adaptive timeouts, whose durations increase when they are trigged within a consensus instance. Considering that the initial timeouts are of 3s and 1s, we realize that the latencies of blocks assigned to crashed validators are essentially dominated by timeouts.

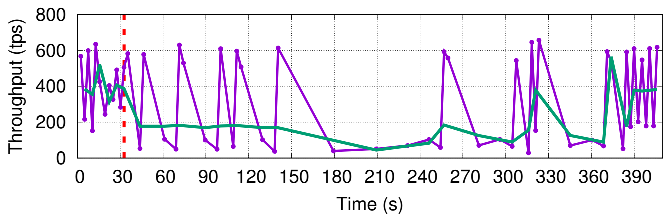

Fig. 10. Instantaneous throughput per block and every two blocks in a network with 128 nodes, 42 of which crashed at the time marked by the dashed line.

The impact of having crashed validators in the Tendermint network is more clearly observed in Figure 10, which depicts the instantaneous throughput before and after the failures. Blocks that required multiple rounds of consensus to be committed, as they were assigned to crashed validators, have higher latencies and notably bring down throughput. Since almost 1/3 of the nodes crashed, sequences of instances and rounds of consensus assigned to faulty proposers are observed (e.g., from 180s to 250s). At the same time, blocks assigned to correct validators are essentially not affected: they present regular latencies and produce peaks of throughput that can last for several blocks (e.g., from 385s to 405s). The overall throughput, however, is severely impacted: 164 tps, 63% lower than in the fault-free experiments (438 tps). The average latency was 9.17 $\pm$ 8.13s, 2.7x higher than in the baseline experiments (3.45 $\pm$ 0.99s), and more disperse. In fact, almost 30% of transactions presented latencies above 10s, and about 12% above 20s.

E. *Byzantine faults*

To evaluate Tendermint under malicious validators, we used the setup considered in Section VII-B, with 128 validator nodes and 1536 clients. The experiment duration was doubled, from 30 to 60 blocks. The Byzantine behavior was introduced by configuring pairs of nodes to use the same validator key. Validators are identified by their public keys, used to select the proposer for each height and round of consensus, and to verify blocks proposed and votes for blocks. It is thus assumed that validator keys are uniquely associated to a node. When nodes share the same validator's private key, distinct blocks can be proposed in the same height and round of consensus. In addition, conflicting votes signed with the same validator key can be issued by different nodes, which may generate forks in the blockchain if one third or more keys are compromised. This misbehavior is a type of equivocation attack [13], [14].

To improve the effectiveness of the attack, validator keys were shared by pairs of nodes that are far from each other in the overlay network. This increases the likelihood of having distinct blocks proposed in the same instance of consensus, as nodes sharing the same validator key should have different sets of transactions in their *mempool*s. In addition, as they should not have (many) peers in common, it is more likely to partition the network among conflicting proposals. This behavior was observed in the experiments, where some instances of consensus did not succeed at the first round, as none of the conflicting proposals received enough votes to be committed. Despite the extension of the attack, however, the blockchain neither halted nor forked in any experiment.

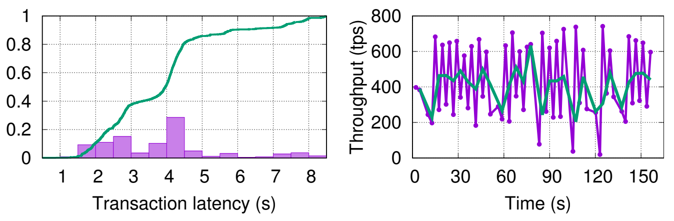

Fig. 11. Distributions of latencies for transactions (left) and instant throughput (right) under Byzantine behavior, with 128 nodes sharing 64 validator keys.

Figure 11 summarizes the performance of Tendermint under Byzantine behavior, with 128 nodes sharing 64 validator keys, each key being shared by two nodes from the system startup. (We also ran experiments with fewer than one-third compromised keys and did not observe any anomalous behavior.) On the left of Figure 11, we present the distribution of latencies for submitted transactions. For about 86% of the transactions, latencies are in line with the results obtained in the fault-free scenario with 64 validators (up to 5s) with average latency of 3.32s (versus 3.07s). In fact, 90% of the blocks were not affected by the Byzantine behavior, being committed in a single round of consensus, with average block latency 2.36s (versus 2.38s). The remaining blocks (10%) were affected by the Byzantine behavior, requiring two rounds of consensus to be committed, with average block latency 5.57s. Observe that the block latency with two rounds is lower than the ones observed in Section VII-D (8.4s), as timeouts were not trigged.

The right part of Figure 11 presents the instant throughput of Tendermint under the considered Byzantine behavior. The troughs are directly related to blocks that required more than one round of consensus to be committed (instants 10, 56, 83, 105, 120, and 137s). The overall throughput in the experiment was 388 tps, 19% lower than in the faulty-free experiments (477 tps). The average latency increased from 3.07s to 3.84s under Byzantine behavior, a 25% degradation. Observe that all comparisons are against the faulty-free scenario with 64 validators, since although we have 128 nodes acting as validators, there are only 64 validators (keys) in the network.

Finally, although this form of Byzantine behavior may seem artificial, it was observed in production Tendermint networks [24]. The reported incident affected a small number of validators, and it resulted from a combination of misconfigured nodes and a software bug. But as an attack vector, cloning keys from legit nodes has potential to be harmful at scale [14], in particular because it does not require coding incorrect behavior.

VIII. Related work

Many blockchain protocols have been proposed in the last few years (e.g, [21], [25]–[30]). Surveying all existing approaches is out of the scope of this paper (and its page limits). Instead, in this section, we overview the blockchain landscape and place Tendermint in this context.

Blockchain systems can be broadly divided into two categories, permissionless and permissioned. Permissionless protocols do not depend on a well-defined set of nodes to execute consensus. In principle, any node can participate in the execution of consensus. Proof-of-work (PoW) blockchain systems, such as Bitcoin [4] and Ethereum [31], are the most prominent representatives of this category of protocols, although many other protocols rely on PoW (e.g., [26], [32]).

The substantial amount of energy consumed by PoW protocols [5], their probabilistic safety guarantees, and their low throughput (i.e, a few transactions per second) and high latency boosted the quest for alternative solutions. In proof-of-stake (PoS) blockchain systems, like Tendermint, nodes are accountable for their acts, which discourages misbehavior.

While Bitcoin maintains a distributed ledger of UTXO transactions that essentially transfer assets, Ethereum provides a Turing complete virtual machine, the Ethereum Virtual Machine (EVM). Users are then allowed to upload code to the EVM in the form of *smart contracts* that are executed by committed transactions. Tendermint also supports arbitrary operations, but it substantially differs from Ethereum not only for the adoption of proof-of-stake, but also for the decision of running a specific application on each Tendermint network. This design can be aligned with the performance and costs requirements of each application.

Permissioned protocols rely on a precise knowledge of the nodes that execute consensus (i.e., validators) and are based on Byzantine agreement [33]. Many agreement protocols have been proposed that can tolerate Byzantine behavior (e.g., [34]–[36]), PBFT [9] being a well-known representative. Permissioned blockchain protocols, like Tendermint and many others (e.g., [27], [28], [37]), perform better than and do not require the computational power of permissionless protocols. As discussed in Section V-B, Tendermint's consensus protocol shares some of PBFT characteristics, without relying on PBFT's complex view change mechanism.

Many blockchain systems have been proposed in the last few years, typically to improve the performance limitations of early designs. PBFT-like protocols rely on 2/3-majority quorums and quadratic communication among peers to reach consensus. To improve performance, some protocols use smaller quorums [25], [37] or reduce communication from quadratic to linear [28], [29]. Moreover, most protocols rely on a leader to reach consensus decisions (e.g., [9], [26], [38]). In large settings, reaching agreement without a leader can help scale performance [25].

IX. Conclusion

This paper presents the design and architecture of Tendermint, a mature system, in the lifespan of blockchains. Besides overviewing Tendermint's main components, we also assessed its performance under different conditions. We draw the following main conclusions from our evaluation. (i) Nodes in the Tendermint network are well-connected (i.e., 60% of the nodes have between 10 and 20 peers, 80% of the nodes are within two hops). (ii) Throughput and latency degraded gracefully with the number of validators (e.g., an $8\times$ increase in system size resulted in an 18% reduction in throughput). (iii) Tendermint's tightly connected network is resilient to crash failures and a type of equivocation attack, in which malicious nodes try to induce honest nodes to misbehave by sending conflicting messages. We attribute this resiliency to the overlay nature of peer-to-peer communication, where malicious nodes cannot easily deceive honest nodes, as in a fully connected network.

Acknowledgments

We wish to thank the anonymous reviewers for the constructive feedback. This work was partially supported by the Swiss National Science Foundation (project number 175717), the Interchain Foundation, and the Hasler Foundation.

References

[1] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[2] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Master's thesis, University of Guelph, Canada, Jun. 2016. [Online]. Available: http://hdl.handle.net/10214/9769

[3] E. Buchman and J. Kwon, "Cosmos whitepaper: a network of distributed ledgers," 2016. [Online]. Available: https://cosmos.network/resources/whitepaper

[4] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," White paper, 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[5] H. Vranken, "Sustainability of bitcoin and blockchains," *Current Opinion in Environmental Sustainability*, vol. 28, pp. 1–9, Oct. 2017.

[6] A. Poelstra, "Distributed consensus from proof of stake is impossible," Self-published Paper, May 2014. [Online]. Available: https://download.wpsoftware.net/bitcoin/old-pos.pdf

[7] V. Buterin, "Slasher: A punitive proof-of-stake algorithm," Jan. 2014. [Online]. Available: https://blog.ethereum.org/2014/01/15/slasher-a-punitive-proof-of-stake-algorithm

[8] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," arXiv:1807.04938 [cs.DC], Jul. 2018. [Online]. Available: https://arxiv.org/abs/1807.04938

[9] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, p. 398461, Nov. 2002.

[10] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, Apr. 1988.

[11] A. Demers, D. Greene, C. Hauser, W. Irish, and J. Larson, "Epidemic algorithms for replicated database maintenance," in *6th annual ACM Symposium on Principles of Distributed Computing*, ser. PODC'87. ACM Press, 1987.

[12] R. Guerraoui, J. Hamza, D. Seredinschi, and M. Vukolic, "Can 100 machines agree?" arXiv:1911.07966 [cs.DC], Nov. 2019. [Online]. Available: http://arxiv.org/abs/1911.07966

[13] A. Clement, F. Junqueira, A. Kate, and R. Rodrigues, "On the (limited) power of non-equivocation," in *PODC*, 2012.

[14] S. Bano, A. Sonnino, A. Chursin, D. Perelman, and D. Malkhi, "Twins: White-glove approach for BFT testing," arXiv:2004.10617 [cs.DC], Apr. 2020. [Online]. Available: https://arxiv.org/abs/2004.10617

[15] M. Jelasity, "Gossip," in *Self-organising Software*, ser. Natural Computing Series. Springer Berlin Heidelberg, 2011, pp. 139–162.

[16] C. E. Bezerra, F. Pedone, and R. V. Renesse, "Scalable state-machine replication," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN'14. IEEE, Jun. 2014.

[17] M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication," in *Open Problems in Network Security*, ser. iNetSec 2015. Springer International Publishing, May 2016, pp. 112–125.

[18] S. Braithwaite, E. Buchman, I. Khoffi, I. Konnov, Z. Milosevic, R. Ruetschi, and J. Widder, "A Tendermint light client," arXiv:2010.07031 [cs.DC], Oct. 2020. [Online]. Available: https://arxiv.org/abs/2010.07031

[19] Y. Amoussou-Guenou, A. D. Pozzo, M. Potop-Butucaru, and S. Tucci-Piergiovanni, "Dissecting tendermint," in *Networked Systems*. Springer, Sep. 2019, pp. 166–182.

[20] Y. Amoussou-Guenou, A. Del Pozzo, M. Potop-Butucaru, and S. Tucci-Piergiovanni, "Correctness of tendermint-core blockchains," in *22nd International Conference on Principles of Distributed Systems*, vol. 125, 2018, pp. 16:1–16:16.

[21] T. Crain, V. Gramoli, M. Larrea, and M. Raynal, "Dbft: Efficient byzantine consensus with a weak coordinator and its application to consortium blockchains," arXiv:1702.03068 [cs.DC], Feb. 2017. [Online]. Available: https://arxiv.org/abs/1702.03068

[22] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *Advances in Cryptology*, ser. CRYPTO 2001. Springer Berlin Heidelberg, Aug. 2001, pp. 524–541.

[23] D. K. Gifford, "Weighted voting for replicated data," in *7th ACM Symposium on Operating Systems Principles*, ser. SOSP '79. New York, NY, USA: Association for Computing Machinery, Dec. 1979, p. 150162.

[24] Iqlusion, "Tendermint KMS-related cosmos hub validator incident," Aug. 2019. [Online]. Available: https://iqlusion.blog/postmortem-2019-08-08-tendermint-kms-related-cosmos-hub-validator-incident

[25] T. Crain, C. Natoli, and V. Gramoli, "Red belly: A secure, fair and scalable open blockchain," in *42nd IEEE Symposium on Security and Privacy*, ser. IEEE SP 2021, May 2021.

[26] I. Eyal, A. E. Gencer, E. G. Sirer, and R. V. Renesse, "Bitcoin-NG: A scalable blockchain protocol," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Mar. 2016, pp. 45–59.

[27] 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. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *13th EuroSys Conference*. ACM, apr 2018.

[28] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: BFT consensus with linearity and responsiveness," in *2019 ACM Symposium on Principles of Distributed Computing*, ser. PODC'19. ACM, Jul. 2019.

[29] ——, "Hotstuff: BFT consensus in the lens of blockchain," arXiv:1803.05069 [cs.DC], Jul. 2019. [Online]. Available: https://arxiv.org/abs/1803.05069

[30] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *26th Symposium on Operating Systems Principles*. ACM, Oct. 2017, pp. 51–68.

[31] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, 2014. [Online]. Available: http://ethereum.github.io/yellowpaper/paper.pdf

[32] Y. Sompolinsky and A. Zohar, "Accelerating bitcoin's transaction processing. fast money grows on trees, not chains," Cryptology ePrint Archive, Report 2013/881, Dec. 2013. [Online]. Available: https://eprint.iacr.org/2013/881

[33] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, Jul. 1982.

[34] 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 Operating Systems Review*, vol. 39, no. 5, pp. 59–74, Oct. 2005.

[35] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," in *21st ACM SIGOPS symposium on Operating systems principles (SOSP '07)*, Oct. 2007, pp. 45–58.

[36] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "Hq replication: A hybrid quorum protocol for byzantine fault tolerance," in *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, vol. 7, Nov. 2006.

[37] R. Guerraoui, F. Huc, and A.-M. Kermarrec, "Highly dynamic distributed computing with byzantine failures," in *2013 ACM symposium on Principles of distributed computing (PODC '13)*, Jul. 2013.

[38] R. Pass and E. Shi, "Thunderella: Blockchains with optimistic instant confirmation," in *EUROCRYPT*, Mar. 2018, pp. 3–33.