ABSTRACT

We present HotStuff, a leader-based Byzantine fault-tolerant replication protocol for the partially synchronous model. Once network communication becomes synchronous, HotStuff enables a correct leader to drive the protocol to consensus at the pace of actual (vs. maximum) network delay—a property called responsiveness—and with communication complexity that is linear in the number of replicas. To our knowledge, HotStuff is the first partially synchronous BFT replication protocol exhibiting these combined properties. Its simplicity enables it to be further pipelined and simplified into a practical, concise protocol for building large-scale replication services.

CCS CONCEPTS

- Software and its engineering → Software fault tolerance;
- Security and privacy → Distributed systems security.

KEYWORDS

Byzantine fault tolerance; consensus; responsiveness; scalability; blockchain

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ACM Reference Format:

1 INTRODUCTION

Byzantine fault tolerance (BFT) refers to the ability of a computing system to endure arbitrary (i.e., Byzantine) failures of its components while taking actions critical to the system’s operation. In the context of state machine replication (SMR) [25, 47], the system as a whole provides a replicated service whose state is mirrored across n deterministic replicas. A BFT SMR protocol is used to ensure that non-faulty replicas agree on an order of execution for client-initiated service commands, despite the efforts of f Byzantine replicas. This, in turn, ensures that the n − f non-faulty replicas will run commands identically and so produce the same response for each command. As is common, we are concerned here with the partially synchronous communication model [25], whereby a known bound Δ on message transmission holds after some unknown global stabilization time (GST). In this model, n ≥ 3f + 1 is required for non-faulty replicas to agree on the same commands in the same order (e.g., [12]) and progress can be ensured deterministically only after GST [27].

When BFT SMR protocols were originally conceived, a typical target system size was n = 4 or n = 7, deployed on a local-area network. However, the renewed interest in Byzantine-fault tolerance brought about by its application to blockchains now demands solutions that can scale to much larger n. In contrast to permissionless blockchains such as the one that supports Bitcoin, for example, so-called permissioned blockchains involve a fixed set of replicas that collectively maintain an ordered ledger of commands or, in other words, that support SMR. Despite their permissioned nature, numbers of replicas in the hundreds or even thousands are envisioned (e.g., [30, 42]). Additionally, their deployment to wide-area networks requires setting Δ to accommodate higher variability in communication delays.

The scaling challenge. Since the introduction of PBFT [20, 21], the first practical BFT replication solution in the partial synchrony model, numerous BFT solutions were built around its core two-phase paradigm. The practical aspect is that a stable leader can drive a consensus decision in just two rounds of message exchanges. The first phase guarantees proposal uniqueness through the formation of a quorum certificate (QC) consisting of (n − f) votes. The second phase guarantees that the next leader can convince replicas to vote for a safe proposal.

The algorithm for a new leader to collect information and propose to replicas—called a view-change—is the epicenter of replication. Unfortunately, view-change based on the two-phase paradigm is far from simple [38], is bug-prone [4], and incurs a significant communication penalty for even moderate system sizes. It requires the new leader to relay information from (n − f) replicas, each reporting its own highest known QC. Even counting just authenticators (digital signatures or message authentication codes), conveying a new proposal has a communication footprint of O(n²) authenticators in PBFT, and variants that combine multiple authenticators into one via threshold digital signatures (e.g., [18, 30]) still send O(n²) authenticators. The total number of authenticators transmitted if O(n) view-changes occur before a single consensus decision is reached is O(n³) in PBFT, and even with threshold signatures is O(n⁵). This scaling challenge plagues not only PBFT, but many other protocols developed since then, e.g., Prime [9], Zyzzyva [34], Upright [22], BFT-SMaRt [13], 700BFT [11], and SBFT [30].

HotStuff revolves around a three-phase core, allowing a new leader to simply pick the highest QC it knows of. It introduces a second phase that allows replicas to "change their mind" after voting in the phase, without requiring a leader proof at all. This alleviates the above complexity, and at the same time considerably simplifies the leader replacement protocol. Last, having (almost)
canonized all the phases, it is very easy to pipeline HotStuff, and to frequently rotate leaders.

To our knowledge, only BFT protocols in the blockchain arena like Tendermint [15, 16] and Casper [17] follow such a simple leader regime. However, these systems are built around a synchronous core, wherein proposals are made in pre-determined intervals that must accommodate the worst-case time it takes to propagate messages over a wide-area peer-to-peer gossip network. In doing so, they forego a hallmark of most practical BFT SMR solutions (including those listed above), namely optimistic responsiveness [42]. Informally, responsiveness requires that a non-faulty leader, once designated, can drive the protocol to consensus in time depending only on the actual message delays, independent of any known upper bound on message transmission delays [10]. More appropriate for our model is optimistic responsiveness, which requires responsiveness only in beneficial (and hopefully common) circumstances—here, after GST is reached. Optimistic or not, responsiveness is precluded with designs such as Tendermint/Casper. The crux of the difficulty is that there may exist an honest replica that has the highest QC, but the leader does not know about it. One can build scenarios where this prevents progress ad infinitum (see Section 4.4 for a detailed liveless scenario). Indeed, failing to incorporate necessary delays at crucial protocol steps can result in losing liveness outright, as has been reported in several existing deployments, e.g., see [2, 3, 19].

Our contributions. To our knowledge, we present the first BFT SMR protocol, called HotStuff, to achieve the following two properties:

- **Linear View Change**: After GST, any correct leader, once designated, sends only $O(n)$ authenticators to drive a consensus decision. This includes the case where a leader is replaced. Consequently, communication costs to reach consensus after GST is $O(n^2)$ authenticators in the worst case of cascading leader failures.

- **Optimistic Responsiveness**: After GST, any correct leader, once designated, needs to wait just for the first $n - f$ responses to guarantee that it can create a proposal that will make progress. This includes the case where a leader is replaced.

Another feature of HotStuff is that the costs for a new leader to drive the protocol to consensus is no greater than that for the current leader. As such, HotStuff supports frequent succession of leaders, which has been argued is useful in blockchain contexts for ensuring chain quality [28].

HotStuff achieves these properties by adding another phase to each view, a small price to latency in return for considerably simplifying the leader replacement protocol. This exchange incurs only the actual network delays, which are typically far smaller than $\Delta$ in practice. As such, we expect this added latency to be much smaller than that incurred by previous protocols that forgo responsiveness to achieve linear view-change. Furthermore, throughput is not affected due to the efficient pipeline we introduce in Section 5.

HotStuff has the additional benefit of being remarkably simple. Safety is specified via voting and commit rules over graphs of nodes. The mechanisms needed to achieve liveness are encapsulated within a Pacemaker, cleanly separated from the mechanisms needed for safety (Section 6).

2 RELATED WORK

Reaching consensus in face of Byzantine failures was formulated as the Byzantine Generals Problem by Lamport et al. [37], who also coined the term “Byzantine failures.” The first synchronous solution was given by Pease et al. [43], and later improved by Dolev and Strong [24]. The improved protocol has $O(n^3)$ communication complexity, which was shown optimal by Dolev and Reischuk [23]. A leader-based synchronous protocol that uses randomness was given by Katz and Koo [32], showing an expected constant-round solution with $(n - 1)/2$ resilience.

Meanwhile, in the asynchronous settings, Fischer et al. [27] showed that the problem is unsolvable deterministically in asynchronous setting in face of a single failure. Furthermore, an $(n - 1)/3$ resilience bound for any asynchronous solution was proven by Ben-Or [12]. Two approaches were devised to circumvent the impossibility. One relies on randomness, initially shown by Ben-Or [12], using independently random coin flips by processes until they happen to converge to consensus. Later works used cryptographic methods to share an unpredictable coin and drive complexities down to constant expected rounds, and $O(n^2)$ communication [18].

The second approach relies on partial synchrony, first shown by Dolev, Lynch, and Stockmeyer (DLS) [25]. This protocol preserves safety during asynchronous periods, and after the system becomes synchronous, DLS guarantees termination. Once synchrony is maintained, DLS incurs $O(n^2)$ total communication and $O(n)$ rounds per decision.

State machine replication relies on consensus at its core to order client requests so that correct replicas execute them in this order. The recurring need for consensus in SMR led Lamport to devise Paxos [36], a protocol that operates an efficient pipeline in which a stable leader drives decisions with linear communication and one round-trip. A similar emphasis led Castro and Liskov [20, 21] to develop an efficient leader-based Byzantine SMR protocol named PBFT, whose stable leader requires $O(n^2)$ communication and two round-trips per decision, and the leader replacement protocol incurs $O(n^3)$ communication. PBFT has been deployed in several systems, including BFT-SMaRt [13]. Kotla et al. introduced an optimistic linear path into PBFT in a protocol named Zyzzyva [34], which was utilized in several systems, e.g., Upright [22] and Byzcoin [33]. The optimistic path has linear complexity, while the leader replacement protocol remains $O(n^3)$. Abraham et al. [4] later exposed a safety violation in Zyzzyva, and presented fixes [5, 30]. On the other hand, to also reduce the complexity of the protocol itself, Song et al. proposed Bosco [49], a simple one-step protocol with low latency on the optimistic path, requiring $5f + 1$ replicas. SBFT [30] introduces an $O(n^2)$ communication view-change protocol that supports a stable leader protocol with optimistically linear, one round-trip decisions. It reduces the communication complexity by harnessing two methods: a collector-based communication paradigm by Reiter [45], and signature combining via threshold cryptography on protocol votes by Cachin et al. [18].

A leader-based Byzantine SMR protocol that employs randomization was presented by Ramasamy et al. [44], and a leaderless
variant named HoneyBadgerBFT was developed by Miller et al. [39]. At their core, these randomized Byzantine solutions employ randomized asynchronous Byzantine consensus, whose best known communication complexity was $O(n^2)$ (see above), amortizing the cost via batching. However, most recently, based on the idea in this HotStuff paper, a parallel submission to PODC’19 [8] further improves the communication complexity to $O(n^2)$.

Bitcoin’s core is a protocol known as Nakamoto Consensus [40], a synchronous protocol with only probabilistic consensus guarantee and no finality (see analysis in [6, 28, 41]). It operates in a permissionless model where participants are unknown, and resilience is kept via Proof-of-Work. As described above, recent blockchain solutions hybridize Proof-of-Work solutions with classical BFT solutions in various ways [7, 17, 26, 29, 31, 33, 42]. The need to address rotating leaders in these hybrid solutions and others provide the motivation behind HotStuff.

### 3 MODEL

We consider a system consisting of a fixed set of $n = 3f + 1$ replicas, indexed by $i \in [n]$ where $[n] = \{1, \ldots, n\}$. A set $F \subseteq [n]$ of up to $f = |F|$ replicas are Byzantine faulty, and the remaining ones are correct. We will often refer to the Byzantine replicas as being coordinated by an adversary, which learns all internal state held by these replicas (including their cryptographic keys, see below).

Network communication is point-to-point, authenticated and reliable: one correct replica receives a message from another correct replica if and only if the latter sent that message to the former. When we refer to a “broadcast”, it involves the broadcaster, if correct, sending the same point-to-point messages to all replicas, including itself. We adopt the partial synchrony model of Dwork et al. [25], where there is a known bound $\Delta$ and an unknown Global Stabilization Time (GST), such that after GST, all transmissions between two correct replicas arrive within time $\Delta$. Our protocol will ensure safety always, and will guarantee progress within a bounded duration after GST. (Guaranteeing progress before GST is impossible [27].) In practice, our protocol will guarantee progress if the system remains stable (i.e., if messages arrive within $\Delta$ time) for sufficiently long after GST, though assuming that it does so forever simplifies discussion.

#### Cryptographic primitives

HotStuff makes use of threshold signatures [14, 18, 48]. In a $(k, n)$-threshold signature scheme, there is a single public key held by all replicas, and each of the $n$ replicas holds a distinct private key. The $i$-th replica can use its private key to contribute a partial signature $\rho_i \leftarrow tsign_i(m)$ on message $m$. Partial signatures $\{\rho_i\}_{i \in [k]}$, where $|k| = k$ and each $\rho_i \leftarrow tsign_i(m)$, can be used to produce a digital signature $\sigma \leftarrow tcombine(m, \{\rho_i\}_{i \in [k]})$ on $m$. Any other replica can verify the signature using the public key and the function $\text{tverify}$. We require that if $\rho_i \leftarrow tsign_i(m)$ for each $i \in I$, $|I| = k$, and if $\sigma \leftarrow tcombine(m, \{\rho_i\}_{i \in [k]})$, then $\text{tverify}(m, \sigma)$ returns true. However, given oracle access to oracles $(tsign_i(m))_{i \in [n]}$, an adversary who queries $tsign_i(m)$ on strictly fewer than $k-f$ of these oracles has negligible probability of producing a signature $\sigma$ for the message $m$ (i.e., such that $\text{tverify}(m, \sigma)$ returns true). Throughout this paper, we use a threshold of $k = 2f + 1$. Again, we will typically leave invocations of $\text{tverify}$ implicit in our protocol descriptions.

We also require a cryptographic hash function $h$ (also called a message digest function), which maps an arbitrary-length input to a fixed-length output. The hash function must be collision resistant [46], which informally requires that the probability of an adversary producing inputs $m$ and $m'$ such that $h(m) = h(m')$ is negligible. As such, $h(m)$ can serve as an identifier for a unique input $m$ in the protocol.

#### Complexity measure

The complexity measure we care about is authenticator complexity, which specifically is the sum, over all replicas $i \in [n]$, of the number of authenticators received by replica $i$ in the protocol to reach a consensus decision after GST. (Again, before GST, a consensus decision might not be reached at all in the worst case [27].) Here, an authenticator is either a partial signature or a signature. Authenticator complexity is a useful measure of communication complexity for several reasons. First, like bit complexity and unlike message complexity, it hides unnecessary details about the transmission topology. For example, $n$ messages carrying one authenticator count the same as one message carrying $n$ authenticators. Second, authenticator complexity is better suited than bit complexity for capturing costs in protocols like ours that reach consensus repeatedly, where each consensus decision (or each view proposed on the way to that consensus decision) is identified by a monotonically increasing counter. That is, because such a counter increases indefinitely, the bit complexity of a protocol that sends such a counter cannot be bounded. Third, since in practice, cryptographic operations to produce or verify digital signatures and to produce or combine partial signatures are typically the most computationally intensive operations in protocols that use them, the authenticator complexity provides insight into the computational burden of the protocol, as well.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Correct leader</th>
<th>Authenticator complexity</th>
<th>Responsiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td></td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td></td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td></td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td></td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td></td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td></td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td></td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

$\sigma$ in the protocol to reach a consensus decision after GST. (Again, before GST, a consensus decision might not be reached at all in the worst case [27].) Here, an authenticator is either a partial signature or a signature. Authenticator complexity is a useful measure of communication complexity for several reasons. First, like bit complexity and unlike message complexity, it hides unnecessary details about the transmission topology. For example, $n$ messages carrying one authenticator count the same as one message carrying $n$ authenticators. Second, authenticator complexity is better suited than bit complexity for capturing costs in protocols like ours that reach consensus repeatedly, where each consensus decision (or each view proposed on the way to that consensus decision) is identified by a monotonically increasing counter. That is, because such a counter increases indefinitely, the bit complexity of a protocol that sends such a counter cannot be bounded. Third, since in practice, cryptographic operations to produce or verify digital signatures and to produce or combine partial signatures are typically the most computationally intensive operations in protocols that use them, the authenticator complexity provides insight into the computational burden of the protocol, as well.

### 4 BASIC HOTSTUFF

HotStuff solves the State Machine Replication (SMR) problem. At the core of SMR is a protocol for deciding on a growing log of command requests by clients. A group of state-machine replicas apply commands in sequence order consistently. A client sends a command request to all replicas, and waits for responses from $(f+1)$ of them. For the most part, we omit the client from the
discussion, and defer to the standard literature for issues regarding numbering and de-duplication of client requests.

The Basic HotStuff solution is presented in Algorithm 2. The protocol works in a succession of views numbered with monotonically increasing view numbers. Each viewNumber has a unique dedicated leader known to all. Each replica stores a tree of pending commands as its local data structure. Each tree node contains a proposed command (or a batch of them), metadata associated with the protocol, and a parent link. The branch led by a given node is the path from the node all the way to the tree root by visiting parent links. During the protocol, a monotonically growing branch becomes committed. To become committed, the leader of a particular view proposing the branch must collect votes from a quorum of \((n - f)\) replicas in three phases, PREPARE, PRE-COMMIT, and COMMIT.

A key ingredient in the protocol is a collection of \((n - f)\) votes over a leader proposal, referred to as a quorum certificate (or "QC" in short). The QC is associated with a particular node and a view number. The combine utility employs a threshold signature scheme to generate a representation of \((n - f)\) signed votes as a single authenticator.

Below we give an operational description of the protocol logic by phases, followed by a precise specification in Algorithm 2, and conclude the section with safety, liveness, and complexity arguments.

4.1 Phases

**PREPARE phase.** The protocol for a new leader starts by collecting new-view messages from \((n - f)\) replicas. The new-view message is sent by a replica as it transitions into viewNumber (including the first view) and carries the highest prepare QC that the replica received (⊥ if none), as described below.

The leader processes these messages in order to select a branch that has the highest preceding view in which a prepare QC was formed. The leader selects the prepare QC with the highest view, denoted high QC, among the new-view messages. Because high QC is the highest among \((n - f)\) replicas, no higher view could have reached a commit decision. The branch led by high QC node is therefore safe.

Collecting new-view messages to select a safe branch may be omitted by an incumbent leader, who may simply select its own highest prepare QC as high QC. We defer this optimization to Section 6 and only describe a single, unified leader protocol in this section. Note that, different from PBFT-like protocols, including this step in the leader protocol is straightforward, and it incurs the same, linear overhead as all the other phases of the protocol, regardless of the situation.

The leader uses the createLeaf method to extend the tail of high QC node with a new proposal. The method creates a new leaf node as a child and embeds a digest of the parent in the child node. The leader then sends the new node in a prepare message to all other replicas. The proposal carries high QC for safety justification.

Upon receiving the prepare message from the current view from the leader, replica \(r\) uses the safeNode predicate to determine whether to accept it. If it is accepted, the replica sends a prepare vote with a partial signature (produced by tsign,\(_r\)) for the proposal to the leader.

**SAFE.NODE predicate.** The safeNode predicate is a core ingredient of the protocol. It examines a proposal message \(m\) carrying a QC justification \(m.justify\), and determines whether \(m.node\) is safe to accept. The safety rule to accept a proposal is the branch of \(m.node\) extends from the currently locked node locked QC node. Additionally, the liveness rule is the replica will accept \(m\) if \(m.justify\) has a higher view than the current locked QC.

**PRE-COMMIT phase.** When the leader receives \((n - f)\) prepare votes for the current proposal curProposal, it combines them into a prepare QC. The leader broadcasts prepare QC in pre-commit messages. A replica responds to the leader with pre-commit vote having a signed digest of the proposal.

**COMMIT phase.** The commit phase is similar to pre-commit phase. When the leader receives \((n - f)\) pre-commit votes, it combines them into a precommit QC and broadcasts it in commit messages; replicas respond to it with a commit vote. Importantly, a replica becomes locked on the precommit QC at this point by setting its locked QC to precommit QC (Line 25 of Algorithm 2). This is crucial to guard the safety of the proposal in case it becomes a consensus decision.

**Decide phase.** When the leader receives \((n - f)\) commit votes, it combines them into a commit QC. Once the leader has assembled a commit QC, it sends it in a decide message to all other replicas. Upon receiving a decide message, a replica considers the proposal embodied in the commit QC a committed decision, and executes the commands in the committed branch. The replica increments viewNumber and starts the next view.

**nextView interrupt.** In all phases, a replica waits for a message at view viewNumber for a timeout period, determined by an auxiliary nextView(viewNumber) utility. If nextView(viewNumber) interrupts waiting, the replica also increments viewNumber and starts the next view.

4.2 Data Structures

**Messages.** A message \(m\) in the protocol has a fixed set of fields that are populated using the Msg() utility shown in Algorithm 1. \(m\) is automatically stamped with curView, the sender’s current view number. Each message has a type \(m.type \in \{\text{NEW-VIEW, PREPARE, PRE-COMMIT, COMMIT, DECIDE}\}\). \(m.node\) contains a proposed node (the leaf node of a proposed branch). There is an optional field \(m.justify\). The leader always uses this field to carry the QC for the different phases. Replicas use it in new-view messages to carry the highest prepare QC. Each message sent in a replica role contains a partial signature \(m.partialSig\) by the sender over the tuple \((m.type, m.viewNumber, m.node)\), which is added in the voteMsg() utility.

**Quorum certificates.** A Quorum Certificate (QC) over a tuple \((type, viewNumber, node)\) is a data type that combines a collection of signatures for the same tuple signed by \((n - f)\) replicas. Given a QC qc, we use qc.type, qc.viewNumber, qc.node to refer to the matching fields of the original tuple.

**Tree and branches.** Each command is wrapped in a node that additionally contains a parent link which could be a hash digest of the parent node. We omit the implementation details from the pseudocode. During the protocol, a replica delivers a message only
after the branch led by the node is already in its local tree. In practice, a recipient who falls behind can catch up by fetching missing nodes from other replicas. For brevity, these details are also omitted from the pseudocode. Two branches are conflicting if neither one is an extension of the other. Two nodes are conflicting if the branches led by them are conflicting.

**Bookkeeping variables.** A replica uses additional local variables for bookkeeping the protocol state: (i) a `viewNumber`, initially 1 and incremented either by finishing a decision or by a `nextView` interrupt; (ii) a locked quorum certificate `locked QC`, initially ⊥, storing the highest QC for which a replica voted `commit`; and (iii) a highest QC `prepare QC`, initially ⊥, storing the highest QC for which a replica voted `pre-commit`. Additionally, in order to incrementally execute a committed log of commands, the replica maintains the highest node whose branch has been executed. This is omitted below for brevity.

### 4.3 Protocol Specification

The protocol given in Algorithm 2 is described as an iterated view-by-view loop. In each view, a replica performs phases in succession based on its role, described as a succession of “as” blocks. A replica can have more than one role. For example, a leader is also a (normal) replica. Execution of `as` blocks across roles can be proceeded concurrently. The execution of each `as` block is atomic. A `nextView` interrupt aborts all operations in any `as` block, and jumps to the “Finally” block.

**Algorithm 1** Utilities (for replica r).

1. function `Msg(type, node, qc)`
2. \[ m \rightarrow \text{type} \]
3. \[ m \rightarrow \text{viewNumber} \rightarrow \text{curView} \]
4. \[ m \rightarrow \text{node} \]
5. \[ m \rightarrow \text{justify} \rightarrow \text{qc} \]
6. return `m`
7. function `voteMsg(type, node, qc)`
8. \[ m \rightarrow \text{Msg(type, node, qc)} \]
9. \[ m \rightarrow \text{partialSig} \rightarrow \text{tmsg, (m, type, viewNumber, node)} \]
10. return `m`
11. procedure `createLeaf(parent, cmd)`
12. \[ b, parent \rightarrow \text{parent} \]
13. \[ b, cmd \rightarrow \text{cmd} \]
14. return `b`
15. function `QC(V)`
16. \[ qc.type \rightarrow \text{type} : m \in V \]
17. \[ qc.viewNumber \rightarrow \text{viewNumber} : m \in V \]
18. \[ qc.node \rightarrow \text{node} : m \in V \]
19. \[ qc.sig \rightarrow \text{tcombine(qc.type, qc.viewNumber, qc.node}, {m, partialSig m e V}) \]
20. return `qc`
21. function `matchMsg(m, t, v)`
22. return `(m.type = t) \land (m.viewNumber = v)`
23. function `matchQC(qc, t, v)`
24. return `(qc.type = t) \land (qc.viewNumber = v)`
25. function `safeNode(qc, node)`
26. return `(node extends from `locked QC, node`) \lor // safety rule
27. `(qc.viewNumber > `locked QC, viewNumber) \lor // liveness rule

**Algorithm 2** Basic HotStuff protocol (for replica r).

1. for `curView` ← 1, 2, 3, . . . do
   2. as a leader // `r = leader(curView)`

   **PRE-COMMIT phase**
   3. wait for `(n - f)` new-view messages from `view 0`
   4. `M ← {m | matchingMsg(m, viewNumber, curView − 1))}
   5. `high QC ← arg max {m, justify, viewNumber} \cdot justify`
   6. `curProposal ← createLeaf(high QC, node, client’s command)`
   7. broadcast `Msg(prepare, curProposal, high QC)`
   8. as a replica
   9. wait for message `m` from `leader(curView)
   10. `M ← matchingMsg(m, prepare, curView)`
   11. `precommit QC ← QC(V)`
   12. broadcast `Msg(precommit, ⊥, precommit QC)`
   13. as a replica
   14. wait for message `m` from `leader(curView)
   15. `M ← matchingQC(m, justify, prepare, curView)`
   16. `commit QC ← m, justify`
   17. `send to leader(curView)`
   18. `voteMsg(precommit, m, justify, node, ⊥)`

   **COMMIT phase**
   19. as a leader
   20. wait for `(n - f)` votes:
   21. `V ← {v | matchingMsg(v, precommit, curView)}
   22. precommit QC ← QC(V)
   23. broadcast `Msg(commit, ⊥, precommit QC)`
   24. as a replica
   25. wait for message `m` from `leader(curView)
   26. `M ← matchingQC(m, justify, precommit, curView)`
   27. `locked QC ← m, justify`
   28. `send to leader(curView)`
   29. `voteMsg(commit, m, justify, node, ⊥)`

   **DECIDE phase**
   30. as a leader
   31. wait for `(n - f)` votes:
   32. `V ← {v | matchingMsg(v, commit, curView)}
   33. `commit QC ← QC(V)
   34. broadcast `Msg(decide, ⊥, commit QC)`
   35. as a replica
   36. wait for message `m` from `leader(curView)
   37. `M ← matchingQC(m, justify, commit, curView)`
   38. execute new commands through `m, justify, node, respond to clients`

   **Finally**
   39. `nextView interrupt: goto this line if nextView(curView) is called during “wait for” in any phase`
   40. `send Msg(new-view, ⊥, precommit QC) to leader(curView + 1)`

### 4.4 Safety, Liveness, and Complexity

**Safety.** We first define a quorum certificate `qc` to be valid if `tverify((qc.type, qc.viewNumber, qc.node), qc.sig)` is true.

**Lemma 1.** For any valid `qc1, qc2` in which `qc1.type = qc2.type` and `qc1.node` conflicts with `qc2.node`, we have `qc1.viewNumber ≠ qc2.viewNumber`.
Proof. To show a contradiction, suppose \( QC_1, \text{viewNumber} = QC_2, \text{viewNumber} = 1 \). Because a valid QC can be formed only with \( n - f = 2f + 1 \) votes (i.e., partial signatures) for it, there must be a correct replica who voted twice in the same phase of \( v \). This is impossible because the pseudocode allows voting only once for each phase in each view.

\[ \square \]

**Theorem 2.** If \( w \) and \( b \) are conflicting nodes, then they cannot be both committed, each by a correct replica.

Proof. We prove this important theorem by contradiction. Let \( QC_1 \) denote a valid commit QC (i.e., \( QC_1, \text{type} = \text{commit} \)) such that \( QC_1, \text{node} = w \), and \( QC_2 \) denote a valid commit QC such that \( QC_2, \text{node} = b \). Denote \( v_1 = QC_1, \text{viewNumber} \) and \( v_2 = QC_2, \text{viewNumber} \). By Lemma 1, \( v_1 \neq v_2 \). W.l.o.g., assume \( v_1 < v_2 \).

We will now denote by \( v_k \) the lowest view higher than \( v_1 \) for which there is a valid prepare QC, \( QC_k \) (i.e., \( QC_k, \text{type} = \text{prepare} \)) where \( QC_k, \text{viewNumber} = v_k \), and \( QC_k, \text{node} \) conflicts with \( w \). Formally, we define the following predicate for any prepare QC:

\[ E(\text{prepare QC}) := (v_1 < \text{prepare QC}. \text{viewNumber} \leq v_2) \]

\[ \wedge (\text{prepare QC}. \text{node} \text{ conflicts with } w). \]

We can now set the first switching point \( QC_s \):

\[ QC_s := \arg \min_{\text{prepare QC}} \left\{ \text{prepare QC}. \text{viewNumber} \mid \text{prepare QC} \text{ is valid} \wedge E(\text{prepare QC}) \right\}. \]

Note that, by assumption such a \( QC_s \) must exist; for example, \( QC_s \) could be the \( \text{prepare QC} \) formed in view \( v_2 \).

Of the correct replicas that sent a partial result \( tsign_r,((QC_1, \text{type}, QC_1, \text{viewNumber}, QC_1, \text{node})) \), let \( r \) be the first that contributed \( tsign_r,((QC_s, \text{type}, QC_s, \text{viewNumber}, QC_s, \text{node})) \); such an \( r \) must exist since otherwise, one of \( QC_s, \text{sig} \) and \( QC_s, \text{sig} \) could not have been created. During view \( v_2 \), replica \( r \) updates its lock \( QC \) to a \( \text{precommit QC} \) on \( w \) at Line 25 of Algorithm 2. Due to the minimality of \( v_2 \), the lock that replica \( r \) has on \( w \) is not changed before \( QC_s \) is formed. Otherwise, \( QC_s \) must have seen some other \( \text{prepare QC} \) with lower view because Line 17 comes before Line 25, contradicting to the minimality. Now consider the invocation of \( \text{safeNode} \) in the \( \text{prepare} \) phase of view \( v_s \) by replica \( r \), with a message \( m \) carrying \( m, \text{node} = QC_s, \text{node} \). By assumption, \( m, \text{node} \) conflicts with \( locked QC, \text{node} \), and so the disjunct at Line 26 of Algorithm 1 is false. Moreover, \( m, \text{justified}, \text{viewNumber} > v_1 \) would violate the minimality of \( v_2 \), and so the disjunct in Line 27 of Algorithm 1 is also false. Thus, \( \text{safeNode} \) must return false and \( r \) cannot cast a \( \text{prepare} \) vote on the conflicting branch in view \( v_s \), a contradiction.

\[ \square \]

**Liveness.** There are two functions left undefined in the previous section: \( \text{leader} \) and \( \text{nextView} \). Their definition will not affect safety of the protocol, though they do matter to liveness. Before giving candidate definitions for them, we first show that after GST, there is a bounded duration \( T_f \) such that if all correct replicas remain in view \( v \) during \( T_f \) and the leader for view \( v \) is correct, then a decision is reached. Below, we say that \( QC_1 \) and \( QC_2 \) match if \( QC_1 \) and \( QC_2 \) are valid, \( QC_1, \text{node} = QC_2, \text{node} \), and \( QC_1, \text{viewNumber} = QC_2, \text{viewNumber} \).

**Lemma 3.** If a correct replica is locked such that \( locked QC = precommit QC \), then at least \( f + 1 \) correct replicas voted for some \( \text{prepare QC} \) matching \( locked QC \).

Proof. Suppose replica \( r \) is locked on \( \text{precommit QC} \). Then, \( (n - f) \) votes were cast for the matching \( \text{prepare QC} \) in the \( \text{prepare} \) phase (Line 10 of Algorithm 2), out of which at least \( f + 1 \) were from correct replicas.

\[ \square \]

**Theorem 4.** After GST, there exists a bounded time period \( T_f \) such that if all correct replicas remain in view \( v \) during \( T_f \) and the leader for view \( v \) is correct, then a decision is reached.

Proof. Starting in a new view, the leader collects \( (n - f) \) new-view messages and calculates its high QC before broadcasting a \( \text{precommit message} \). Suppose among all replicas (including the leader itself), the highest kept lock is \( locked QC = precommit QC \). By Lemma 3, we know there are at least \( f + 1 \) correct replicas that voted for a \( \text{prepare QC} \) matching \( precommit QC \), and have already sent them to the leader in their new-view messages. Thus, the leader must learn a matching \( \text{prepare QC} \) in at least one of these new-view messages and use it as \( highQC \) in its \( \text{precommit message} \). By the assumption, all correct replicas are synchronized in their view and the leader is non-faulty. Therefore, all correct replicas will vote in the \( \text{prepare} \) phase, since in \( \text{safeNode} \), the condition on Line 27 of Algorithm 1 is satisfied (even if the \( \text{node} \) in the message conflicts with a replica’s stale \( locked QC, \text{node} \), and so Line 26 is not). Then, after the leader assembles a valid \( \text{prepare QC} \) for this view, all replicas will vote in all the following phases, leading to a new decision. After GST, the duration \( T_f \) for these phases to complete is of bounded length.

The protocol (Optimistically) Responsive because there is no explicit "wait-for-\( \Delta \)" step, and the logical disjunction in \( \text{safeNode} \) is used to override a stale lock with the help of the three-phase paradigm.

\[ \square \]

We now provide simple constructions for \( \text{leader} \) and \( \text{nextView} \) that suffice to ensure that after GST, eventually a view will be reached in which the leader is correct and all correct replicas remain in this view for \( T_f \) time. It suffices for \( \text{leader} \) to return some deterministic mapping from view number to a replica, eventually rotating through all replicas. A possible solution for \( \text{nextView} \) is to utilize an exponential back-off mechanism that maintains a timeout interval. Then a timer is set upon entering each view. When the timer goes off without making any decision, the replica doubles the interval and calls \( \text{nextView} \) to advance the view. Since the interval is doubled at each time, the waiting intervals of all correct replicas will eventually have at least \( T_f \) overlap in common, during which the leader could drive a decision.

**Liveliness with two-phases.** We now briefly demonstrate an infinite non-deciding scenario for a "two-phase" HotStuff. This explains the necessity for introducing a synchronous delay in Casper and Tendermint, and hence for abandoning (Optimistic) Responsiveness.

In the two-phase HotStuff variant, we omit the \( \text{pre-commit} \) phase and proceed directly to \( \text{commit} \). A replica becomes locked when it votes on a \( \text{prepare QC} \). Suppose, in view \( v \), a leader proposes \( b \). It completes the \( \text{prepare phase} \), and some replica \( r_1 \) votes
for the \textit{prepare QC}, say \textit{qc}, such that \textit{qc} node = \textit{b}. Hence, \textit{r}_0 becomes locked on \textit{qc}. An asynchronous network scheduling causes the rest of the replicas to move to view \textit{v} + 1 without receiving \textit{qc}.

We now repeat \textit{ad infinitum} the following single-view transcript. We start view \textit{v} + 1 with only \textit{r}_0, holding the highest \textit{prepare QC} (i.e., \textit{qc}) in the system. The new leader \textit{l} collects new-view messages from 2\textit{f} + 1 replicas excluding \textit{r}_0. The highest \textit{prepare QC} among these, \textit{qc}’, has view \textit{v} − 1 and \textit{b’} = \textit{qc’}.node conflicts with \textit{b}. \textit{l} then proposes \textit{b”}, which extends \textit{b’}, to which \textit{b} honest replicas respond with a vote, but \textit{r}_0 rejects it because it is locked on \textit{qc}, \textit{b”} conflicts with \textit{b} and \textit{qc’} is lower than \textit{qc}. Eventually, 2\textit{f} replicas give up and move to the next view. Just then, a faulty replica responds to \textit{l}’s proposal, \textit{l} then puts together a \textit{prepare QC}(\textit{v} + 1, \textit{b”}) and one replica, say \textit{r}_{\textit{v}+1} votes for it and becomes locked on it.

\textbf{Complexity.} In each phase of HotStuff, only the leader broadcasts to all replicas while the replicas respond to the sender once with a partial signature to certify the vote. In the leader’s message, the QC consists of a proof of \((n−f)\) votes collected previously, which can be encoded by a single threshold signature. In a replica’s response, the partial signature from that replica is the only authenticator. Therefore, in each phase, there are \(O(n)\) authenticators received in total. As there is a constant number of phases, the overall complexity per view is \(O(n)\).

5 \textbf{CHAINED HOTSTUFF}

It takes three phases for a Basic HotStuff leader to commit a proposal. These phases are not doing “useful” work except collecting votes from replicas, and they are all very similar. In Chained HotStuff, we improve the Basic Hotstuff protocol utility while at the same time considerably simplifying it. The idea is to change the view on every \textit{prepare phase}, so each proposal has its own view. This reduces the number of message types and allows for pipelining of decisions. A similar approach for message type reduction was suggested in Casper [1].

More specifically, in Chained Hotstuff the votes over a \textit{prepare} phase are collected in a view by the leader into a \textit{prepare QC}. Then the \textit{prepare QC} is relayed to the leader of the next view, essentially delegating responsibility for the next phase, which would have been \textit{pre-commit}, to the next leader. However, the next leader does not actually carry a \textit{pre-commit} phase, but instead initiates a new \textit{prepare} phase and adds its own proposal. This \textit{prepare phase} for view \textit{v} + 1 simultaneously serves as the \textit{prepare phase} for view \textit{v}. The \textit{prepare phase} for view \textit{v} + 2 simultaneously serves as the \textit{pre-commit phase} for view \textit{v} + 1 and as the \textit{commit phase} for view \textit{v}. This is possible because all the phases have identical structure.

The pipeline of Basic Hotstuff protocol phases embedded in a chain of Chained Hotstuff proposals is depicted in Figure 1. Views \textit{v}_1, \textit{v}_2, \textit{v}_3 of Chained Hotstuff serve as the \textit{prepare phase}, \textit{pre-commit}, and \textit{commit Basic Hotstuff phases for cmd1} proposed in \textit{v}_1. This command becomes committed by the end of \textit{v}_3: Views \textit{v}_2, \textit{v}_3, \textit{v}_4 serve as the three Basic Hotstuff phases for \textit{cmd2} proposed in \textit{v}_2, and it becomes committed by the end of \textit{v}_4. Additional proposals generated in these phases continue the pipeline similarly, and are denoted by dashed boxes. In Figure 1, a single arrow denotes the \textit{b.parent} field for a node \textit{b}, and a double arrow denotes \textit{b.justify.node}.

Hence, there are only two types of messages in Chained Hotstuff, a \textit{new-view} message and \textit{generic-phase} \textit{generic message}. The \textit{generic QC} functions in all logically pipelined phases. We next explain the mechanisms in the pipeline to take care of locking and committing, which occur only in the \textit{commit} and \textit{decide} phases of Basic Hotstuff.

\textbf{Dummy nodes.} The \textit{prepare QC} used by a leader in some view \textit{viewNumber} may not directly reference the proposal of the preceding view (\textit{viewNumber} − 1). The reason is that the leader of a preceding view fails to obtain a QC, either because there are conflicting proposals, or due to a benign crash. To simplify the tree structure, \textit{createLeaf} extends \textit{prepare QC} node with blank nodes up to the height (the number of parent links on a node’s branch) of the proposing view, so view-numbers are equated with node heights. As a result, the QC embedded in a node \textit{b} may not refer to its parent, i.e., \textit{b.justify.node} may not equal \textit{b.parent} (the last node in Figure 2).

\textbf{One-Chain, Two-Chain, and Three-Chain.} When a node \textit{b} carries a QC that refers to a direct parent, i.e., \textit{b.justify.node} = \textit{b.parent}, we say that it forms a One-Chain. Denote by \textit{b”} = \textit{b.justify.node}. Node \textit{b”} forms a Two-Chain, if in addition to forming a One-Chain, \textit{b”’}, \textit{justify.node} = \textit{b’’’.parent}. It forms a Three-Chain, if \textit{b’’’} forms a Two-Chain.

Looking at chain \textit{b} = \textit{b’’’justify.node}, \textit{b’’’} = \textit{b’’justify.node}, \textit{b”} = \textit{b’justify.node}, ancestry gaps might occur at any one of the nodes. These situations are similar to a leader of Basic Hotstuff failing to complete any one of three phases, and getting interrupted to the next view by \textit{nextView}.

If \textit{b”} forms a One-Chain, the \textit{prepare phase} of \textit{b”} has succeeded. Hence, when a replica votes for \textit{b”}, it should remember \textit{prepare QC} \leftarrow \textit{b’’’justify.node}. We remark that it is safe to update \textit{prepare QC} even when a One-Chain is not direct, so long as it is higher than the current \textit{prepare QC}. In the implementation code described in Section 6, we indeed update \textit{prepare QC} in this case.

If \textit{b”} forms a Two-Chain, then the \textit{pre-commit phase} of \textit{b”} has succeeded. The replica should therefore update \textit{locked QC} \leftarrow \textit{b’’’justify.node}. Again, we remark that the lock can be updated even when a Two-Chain is not direct—safety will not break—and indeed, this is given in the implementation code in Section 6.

Finally, if \textit{b”} forms a Three-Chain, the \textit{commit phase} of \textit{b} has succeeded, and \textit{b} becomes a committed decision.

Algorithm 3 shows the pseudocode for Chained Hotstuff. The proof of safety given by [50] is similar to the one for Basic Hotstuff. We require the QC in a valid node refers to its ancestor. For brevity, we assume the constraint always holds and omit checking in the code.

\begin{algorithm}
\caption{Chained Hotstuff protocol.}
1: \textbf{procedure} \textit{createLeaf}(parent, cmd, qc)  
2: \hspace{1em} \textit{b.parent} \leftarrow \text{branch extending with blanks from parent} to height \textit{curView}; \textit{b.cmd} \leftarrow cmd; \textit{b.justify} \leftarrow qc; \textbf{return} \textit{b}
\end{algorithm}

1: \textbf{for} \textit{curView} \leftarrow 1, 2, 3, \ldots \textbf{do}
2: \hspace{1em} \textbf{\&\& generic phase}
3: \hspace{2em} \textit{as a leader} // \textit{r} = \text{leader(curView)}
4: \hspace{2em} \text{wait for} \((n-f)\) \textit{new-view messages}:
5: \hspace{3em} \textit{M} \leftarrow \{\textit{M} \text{ matchingMsg}(\text{new-view, curView-\textit{m})}} \text{ // \textit{M} includes the previous leader new-view message, if received
Because of its simplicity, we can easily turn Algorithm 3 into an event-driven-style specification that is almost like the code skeleton for a prototype implementation.

As shown in Algorithm 4, the code is further simplified and generalized by extracting the liveness mechanism from the body into a module named Pacemaker. Instead of the next leader always waiting for a prepare QC at the end of the generic phase before starting its reign, this logic is delegated to the Pacemaker. A stable leader can skip this step and streamline proposals across multiple heights. Additionally, we relax the direct parent constraint for maintaining the highest prepare QC and locked QC, while still preserving the requirement that the QC in a valid node always refers to its ancestor. The proof of correctness is similar to Chained HotStuff and we also defer it to the appendix of [50].

**Data structures.** Each replica $u$ keeps track of the following main state variables:

- $V[\cdot]$ mapping from a node to its votes.
- $\vheight$ height of last voted node.
- $b_{\text{lock}}$ locked node (similar to locked QC).
- $b_{\text{exec}}$ last executed node.
- $q_{\text{high}}$ highest known QC (similar to prepare QC) kept by a Pacemaker.
- $b_{\text{leaf}}$ leaf node kept by a Pacemaker.

It also keeps a constant $b_0$, the same genesis node known by all correct replicas. To bootstrap, it contains a hard-coded QC for itself, and $b_{\text{lock}}, b_{\text{exec}}, b_{\text{leaf}}$ are all initialized to $b_0$.

**Pacemaker.** A Pacemaker is a mechanism that guarantees progress after GST. It achieves this through two ingredients. The first one is "synchronization", bringing all correct replicas, and a unique leader, into a common height for a sufficiently long period. The usual synchronization mechanism in the literature [15, 20, 25] is for replicas to increase the count of a’s they spend at larger heights, until progress is being made. A common way to deterministically elect a leader is to use a rotating leader scheme in which all correct replicas keep a predefined leader schedule and rotate to the next one when the leader is demoted.
Second, a Pacemaker needs to provide the leader with a way to choose a proposal that will be supported by correct replicas. As shown in Algorithm 5, after a view change, in onReceiveNewView, the new leader collects new-message messages sent by replicas through onNextSyncView to discover the highest QC to satisfy the second part of the condition in onReceiveProposal for liveness (Line 18 of Algorithm 4). During the same view, however, the incumbent leader will chain the new node to the end of the leaf last proposed by itself, where no new-message message or update to \(qc'_{high}\) is needed. Based on some application-specific heuristics (to wait until the previously proposed node gets a QC, for example), the current leader invokes onBeat to propose a new node carrying the command to be executed.

It is worth noting that even if a bad Pacemaker invokes onPropose arbitrarily, or selects a parent and a QC capriciously, and against any scheduling delays, safety is always guaranteed. Therefore, safety guaranteed by Algorithm 4 alone is entirely decoupled from liveness by any potential instantiation of Algorithm 5.

Algorithm 4 Event-driven HotStuff (for replica \(u\)).

```
1: procedure createLeaf(parent, cmd, qc, height)
2: \(b_{parent} \leftarrow \text{parent}; b_{cmd} \leftarrow \text{cmd};\)
3: \(b_{justify} \leftarrow \text{qc}; b_{height} \leftarrow \text{height}; \text{return } b\)
4: procedure update(b’)
5: \(b' \leftarrow b_{justify}\text{ node; } b' \leftarrow b'_{justify}\text{ node}\)
6: // PRE-COMMIT phase on \(b'\)
7: updateQCHigh(b’, justify)
8: if \(b’.height > b_{lock}.height\) then
9: \(b_{lock} \leftarrow b’\); // COMMIT phase on \(b’\)
10: if \((b’.parent = b) \wedge (b, parent = b)\) then
11: onCommit(b)
12: b_{exec} \leftarrow b // DECIDE phase on b
13: procedure onCommit(b)
14: if \(b_{exec}.height < b.height\) then
15: onCommit(b, parent); execute(b, cmd)
16: procedure onReceiveProposal(Msg, (generic, \(b_{new}, \bot\))
17: if \(b_{new}.height > b_{height}\) ∧ \((b_{new} \text{ extends block})\)
18: \(b_{justify}.node, height > b_{lock}.height\) then
19: \(b_{lock} \leftarrow b_{new}.height\)
20: send(getLeader(),voteMsg\((\text{generic, } b_{new}, \bot)\))
21: update(b_{new})
22: procedure onReceiveVote(m = voteMsg\((\text{generic, } b, \bot)\))
23: if \(3(\sigma, \sigma') \in V[b]\) then return // avoid duplicates
24: \(V[b] \leftarrow V[b] \cup \{(\sigma, m, \text{partialSig})\}\) // collect votes
25: if \(V[b] \geq n - f\) then
26: \(qc \leftarrow QC(\{\sigma | (\sigma', \sigma) \in V[b]\})\)
27: updateQCHigh(qc)
28: function onPropose(b_{leaf}, cmd, qc, high)
29: \(b_{leaf} \leftarrow \text{createLeaf}(b_{leaf}, \text{cmd, } qc, high, b_{leaf}.height + 1)\)
// send to all replicas, including \(u\) itself
30: broadcast(Msg\((\text{generic, } b_{new}, \bot)\))
31: return b_{new}
```

Algorithm 5 Code skeleton for a Pacemaker (for replica \(u\)).

```
// We assume Pacemaker in all correct replicas will have synchronized leadership after GST.
1: function getLeader // . . . specified by the application
2: procedure updateQCHigh(qc_{high})
```

Algorithm 6 update replacement for two-phase HotStuff.

```
1: procedure update(b+)
2: \(b' \leftarrow b_{justify}\text{ node; } b' \leftarrow b_{justify}\text{ node}\)
3: updateQCHigh(b’_{justify})
4: if \(b'.height > b_{lock}.height\) then
5: \(b_{lock} \leftarrow b’\)
6: if \((b = b’.parent)\) then onCommit(b); \(b_{exec} \leftarrow b\)
```

Two-phase HotStuff variant. To further demonstrate the flexibility of the HotStuff framework, Algorithm 6 shows the two-phase variant of HotStuff. Only the update procedure is affected, a Two-Chain is required for reaching a commit decision, and a One-Chain determines the lock. As discussed above (Section 4.4), this two-phase variant loses Optimistic Responsiveness, and is similar to Tendermint/Casper. The benefit is fewer phases, while liveness may be addressed by incorporating in Pacemaker a wait based on maximum network delay.

Evaluation. Due to the space limitation, we defer our evaluation details to the longer paper [50]. There, we compare our implementation to BFT-SMaRt, a state-of-the-art implementation based on a two-phase PBFT variant. We show that even though three-phase HotStuff has an additional phase for its responsiveness and uses digital signatures universally (where BFT-SMaRt only uses MACs for votes), it still achieves similar latency, while being able to outperform BFT-SMaRt in throughput. It also scales better than BFT-SMaRt.

ACKNOWLEDGMENTS

We are thankful to Mathieu Baudet, Avery Ching, George Danezis, François Garillot, Zekun Li, Ben Maurer, Kartik Nayak, Dmitri Perelman, and Ling Ren, for many deep discussions of HotStuff, and to Mathieu Baudet for exposing a subtle error in a previous version posted to the ArXiv of this manuscript.

REFERENCES
