HotStuff: BFT Consensus in the Lens of Blockchain

Maofan Yin[1,2], Dahlia Malkhi[2], Michael K. Reiter[2,3], Guy Golan Gueta[2], and Ittai Abraham[2]

[1]Cornell University, [2]VMware Research, [3]UNC-Chapel Hill

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. HotStuff is built around a novel framework that forms a bridge between classical BFT foundations and blockchains. It allows the expression of other known protocols (DLS, PBFT, Tendermint, Casper), and ours, in a common framework.

Our deployment of HotStuff over a network with over 100 replicas achieves throughput and latency comparable to that of BFT-SMaRt, while enjoying linear communication footprint during leader failover (vs. cubic with BFT-SMaRt).

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) [35, 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, $n \geq 3f + 1$ 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., [42, 30]). 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], 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 it 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 $O(n^3)$ authenticators in PBFT, and variants that combine multiple authenticators into one via threshold digital signatures (e.g., [18, 30]) still send $O(n^2)$ authenticators. The total number of authenticators transmitted if $O(n)$ view-changes occur before a single consensus decision is reached is $O(n^4)$ in PBFT, and even with threshold signatures is $O(n^3)$. 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 [3, 2, 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 Δ 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.

In addition to the theoretical contribution, HotStuff also provides insights in understanding BFT replication in general and instantiating the protocol in practice (see Section 6):

- A framework for BFT replication over graphs of nodes. Safety is specified via voting and commit graph rules. Liveness is specified separately via a Pacemaker that extends the graph with new nodes.

- A casting of several known protocols (DLS, PBFT, Tendermint, and Casper) and one new (ours, HotStuff), in this framework.

HotStuff has the additional benefit of being remarkably simple, owing in part to its economy of mechanism: There are only two message types and simple rules to determine how a replica treats each. 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. At the same time, it is expressive in that it allows the representation of several known protocols (DLS, PBFT, Tendermint, and Casper) as minor variations. In part this flexibility derives from its operation over a graph of nodes, in a way that forms a bridge between classical BFT foundations and modern blockchains.

We describe a prototype implementation and a preliminary evaluation of HotStuff. Deployed over a network with over a hundred replicas, HotStuff achieves throughput and latency comparable to, and sometimes exceeding, those of mature systems such as BFT-SMaRt, whose code complexity far exceeds that of HotStuff. We further demonstrate that the communication footprint of HotStuff remains constant in face of frequent leader replacements, whereas BFT-SMaRt grows quadratically with the number of replicas.

Protocol	Authenticator complexity			Responsiveness
	Correct leader	*Leader failure (view-change)*	*f leader failures*	
DLS [25]	$O(n^4)$	$O(n^4)$	$O(n^4)$	
PBFT [20]	$O(n^2)$	$O(n^3)$	$O(fn^3)$	✓
SBFT [30]	$O(n)$	$O(n^2)$	$O(fn^2)$	✓
Tendermint [15] / Casper [17]	$O(n^2)$	$O(n^2)$	$O(fn^2)$	
Tendermint* / Casper*	$O(n)$	$O(n)$	$O(fn)$	
HotStuff	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$	$\boldsymbol{O(fn)}$	**✓**

*Signatures can be combined using threshold signatures, though this optimization is not mentioned in their original works.

Table 1: Performance of selected protocols after GST.

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 $O(n^3)$ communication [18].

The second approach relies on partial synchrony, first shown by Dwork, 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 $O(n^4)$ 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 $O(n^3)$ (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 safety guarantee and no finality (see analysis in [28, 41, 6]). 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 [26, 33, 7, 17, 29, 31, 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 $F \subset [n]$ 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 Δ and an unknown Global Stabilization Time (GST), such that after GST, all transmissions between two correct replicas arrive within time Δ. 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 Δ time) for sufficiently long after GST, though assuming that it does so forever simplifies discussion.

Cryptographic primitives. HotStuff makes use of threshold signatures [48, 18, 14]. 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* $\rho_i \leftarrow \mathit{tsign}_i(m)$ on message m. Partial signatures $\{\rho_i\}_{i\in I}$, where $|I| = k$ and each $\rho_i \leftarrow \mathit{tsign}_i(m)$, can be used to produce a digital signature $\sigma \leftarrow \mathit{tcombine}(m, \{\rho_i\}_{i\in I})$ on m. Any other replica can verify the signature using the public key and the function $\mathit{tverify}$. We require that if $\rho_i \leftarrow \mathit{tsign}_i(m)$ for each $i \in I$, $|I| = k$, and if $\sigma \leftarrow \mathit{tcombine}(m, \{\rho_i\}_{i\in I})$, then $\mathit{tverify}(m, \sigma)$ returns true. However, given oracle access to oracles $\{\mathit{tsign}_i(\cdot)\}_{i\in[n]\setminus F}$, an adversary who queries $\mathit{tsign}_i(m)$ on strictly fewer than $k - f$ of these oracles has negligible probability of producing a signature σ for the message m (i.e., such that $\mathit{tverify}(m, \sigma)$ returns true). Throughout this paper, we use a threshold of $k = 2f + 1$. Again, we will typically leave invocations of $\mathit{tverify}$ 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.

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 $tcombine$ 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 $prepareQC$ that the replica received ($\perp$ 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 $prepareQC$ was formed. The leader selects the $prepareQC$ with the highest view, denoted $highQC$, among the NEW-VIEW messages. Because $highQC$ is the highest among $(n-f)$ replicas, no higher view could have reached a commit decision. The branch led by $highQC.node$ 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 $prepareQC$ as $highQC$. 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 $highQC.node$ 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 $highQC$ for safety justification.

Upon receiving the PREPARE message for 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.

SAFENODE 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 $lockedQC.node$. On the other hand, the liveness rule is the replica will accept m if $m.justify$ has a higher view than the current $lockedQC$. The predicate is true as long as either one of two rules holds.

PRE-COMMIT phase. When the leader receives $(n-f)$ PREPARE votes for the current proposal $curProposal$, it combines them into a $prepareQC$. The leader broadcasts $prepareQC$ 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 $precommitQC$ and broadcasts it in COMMIT messages; replicas respond to it with a COMMIT vote. Importantly, a replica becomes *locked* on the $precommitQC$ at this point by setting its $lockedQC$ to $precommitQC$ (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 $commitQC$. Once the leader has assembled a $commitQC$, it sends it in a decide message to all other replicas. Upon receiving a decide message, a replica considers the proposal embodied in the $commitQC$ 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 $m.type \in \{$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 $prepareQC$. Each message sent in a replica role contains a partial signature $m.partialSig$ by the sender over the tuple $\langle m.type, m.viewNumber, m.node\rangle$, which is added in the voteMsg() utility.

Quorum certificates. A Quorum Certificate (QC) over a tuple $\langle type, viewNumber, node\rangle$ 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 $lockedQC$, initially $\perp$, storing the highest QC for which a replica voted commit; and (iii) a $prepareQC$, initially $\perp$, 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).

```
function Msg($type, node, qc$)
    $m.type \leftarrow type$
    $m.viewNumber \leftarrow curView$
    $m.node \leftarrow node$
    $m.justify \leftarrow qc$
    return $m$
function voteMsg($type, node, qc$)
    $m \leftarrow$ Msg($type, node, qc$)
    $m.partialSig \leftarrow tsign_r(\langle m.type, m.viewNumber, m.node\rangle)$
    return $m$
procedure createLeaf($parent, cmd$)
    $b.parent \leftarrow parent$
    $b.cmd \leftarrow cmd$
    return $b$
function QC($V$)
    $qc.type \leftarrow m.type : m \in V$
    $qc.viewNumber \leftarrow m.viewNumber : m \in V$
    $qc.node \leftarrow m.node : m \in V$
    $qc.sig \leftarrow tcombine(\langle qc.type, qc.viewNumber, qc.node \rangle, \{m.partialSig \mid m \in V\})$
    return $qc$
function MatchingMsg($m, t, v$)
    return $(m.type = t) \wedge (m.viewNumber = v)$
function MatchingQC($qc, t, v$)
    return $(qc.type = t) \wedge (qc.viewNumber = v)$
function SafeNode($node, qc$)
    return ($node$ extends from $lockedQC.node$) $\vee$   // safety rule
   $(qc.viewNumber > lockedQC.viewNumber)$   // liveness rule
```

Algorithm 2 Basic HotStuff protocol (for replica r).

```
for $curView \leftarrow 1, 2, 3, \ldots$ do
  ▷ PREPARE phase
    as a leader   // $r =$ LEADER($curView$)
    // we assume special NEW-VIEW messages from view 0
        wait for $(n - f)$ NEW-VIEW messages: $M \leftarrow \{m \mid$ MatchingMsg($m$, NEW-VIEW, $curView - 1$)$\}$
        $highQC \leftarrow \left( \arg\max_{m \in M} \{m.justify.viewNumber\} \right).justify$
        $curProposal \leftarrow$ CreateLeaf($highQC.node$, client's command)
        broadcast Msg(PREPARE, $curProposal$, $highQC$)
    as a replica
        wait for message $m$ : MatchingMsg($m$, PREPARE, $curView$) from LEADER($curView$)
        if $m.node$ extends from $m.justify.node$ $\wedge$
                    SafeNode($m.node, m.justify$) then
            send voteMsg(PREPARE, $m.node$, $\perp$) to LEADER($curView$)
  ▷ PRE-COMMIT phase
    as a leader
        wait for $(n - f)$ votes: $V \leftarrow \{v \mid$ MatchingMsg($v$, PREPARE, $curView$)$\}$
        $prepareQC \leftarrow$ QC($V$)
        broadcast Msg(PRE-COMMIT, $\perp$, $prepareQC$)
    as a replica
        wait for message $m$ : MatchingQC($m.justify$, PREPARE, $curView$) from LEADER($curView$)
        $prepareQC \leftarrow m.justify$
        send voteMsg(PRE-COMMIT, $m.justify.node$, $\perp$) to LEADER($curView$)
  ▷ COMMIT phase
    as a leader
        wait for $(n - f)$ votes: $V \leftarrow \{v \mid$ MatchingMsg($v$, PRE-COMMIT, $curView$)$\}$
        $precommitQC \leftarrow$ QC($V$)
        broadcast Msg(COMMIT, $\perp$, $precommitQC$)
    as a replica
        wait for message $m$ : MatchingQC($m.justify$, PRE-COMMIT, $curView$) from LEADER($curView$)
        $lockedQC \leftarrow m.justify$
        send voteMsg(COMMIT, $m.justify.node$, $\perp$) to LEADER($curView$)
  ▷ DECIDE phase
    as a leader
        wait for $(n - f)$ votes: $V \leftarrow \{v \mid$ MatchingMsg($v$, COMMIT, $curView$)$\}$
        $commitQC \leftarrow$ QC($V$)
        broadcast Msg(DECIDE, $\perp$, $commitQC$)
    as a replica
        wait for message $m$ from LEADER($curView$)
        wait for message $m$ : MatchingQC($m.justify$, COMMIT, $curView$) from LEADER($curView$)
```

```
34:         execute new commands through $m.justify.node$, respond to clients
    ▷ Finally
35:     NEXTVIEW interrupt: goto this line if NEXTVIEW($curView$) is called during “wait for” in any phase
36:     send MSG(NEW-VIEW, $\bot$, $prepareQC$) to LEADER($curView + 1$)
```

4.4 Safety, Liveness, and Complexity

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

Lemma 1. *For any valid qc_1, qc_2 in which $qc_1.type = qc_2.type$ and $qc_1.node$ conflicts with $qc_2.node$, we have $qc_1.viewNumber \neq qc_2.viewNumber$.*

Proof. To show a contradiction, suppose $qc_1.viewNumber = qc_2.viewNumber = v$. 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. □

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 $commitQC$ (i.e., $qc_1.type =$ COMMIT) such that $qc_1.node = w$, and qc_2 denote a valid $commitQC$ such that $qc_2.node = b$. Denote $v_1 = qc_1.viewNumber$ and $v_2 = qc_2.viewNumber$. By Lemma 1, $v_1 \neq v_2$. W.l.o.g. assume $v_1 < v_2$.

We will now denote by v_s the lowest view higher than v_1 for which there is a valid $prepareQC$, qc_s (i.e., $qc_s.type =$ PREPARE) where $qc_s.viewNumber = v_s$, and $qc_s.node$ conflicts with w. Formally, we define the following predicate for any $prepareQC$:

$$E(prepareQC) := (v_1 < prepareQC.viewNumber \leq v_2) \wedge (prepareQC.node \text{ conflicts with } w).$$

We can now set the *first* switching point qc_s:

$$qc_s := \underset{prepareQC}{\arg\min} \left\{ prepareQC.viewNumber \mid prepareQC \text{ is valid} \wedge E(prepareQC) \right\}.$$

Note that, by assumption such a qc_s must exist; for example, qc_s could be the $prepareQC$ formed in view v_2.

Of the correct replicas that sent a partial result $tsign_r(\langle qc_1.type, qc_1.viewNumber, qc_1.node\rangle)$, let r be the first that contributed $tsign_r(\langle qc_s.type, qc_s.viewNumber, qc_s.node\rangle)$; such an r must exist since otherwise, one of $qc_1.sig$ and $qc_s.sig$ could not have been created. During view v_1, replica r updates its lock $lockedQC$ to a $precommitQC$ on w at Line 25 of Algorithm 2. Due to the minimality of v_s, the lock that replica r has on the branch led by w is not changed before qc_s is formed. Otherwise r must have seen some other $prepareQC$ with lower view because Line 17 comes before Line 25, contradicting to the minimality. Now consider the invocation of SAFENODE in the PREPARE phase of view v_s by replica r, with a message m carrying $m.node = qc_s.node$. By assumption, $m.node$ conflicts with $lockedQC.node$, and so the disjunct at Line 26 of Algorithm 1 is false. Moreover, $m.justify.viewNumber > v_1$ would violate the minimality of v_s, and so the disjunct in Line 27 of Algorithm 1 is also false. Thus, SAFENODE must return false and r cannot cast a PREPARE vote on the conflicting branch in view v_s, a contradiction. □

Liveness. There are two functions left undefined in the previous section: LEADER and 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, $qc_1.node = qc_2.node$, and $qc_1.viewNumber = qc_2.viewNumber$.

Lemma 3. *If a correct replica is locked such that $lockedQC = precommitQC$, then at least $f + 1$ correct replicas voted for some $prepareQC$ matching $lockedQC$.*

Proof. Suppose replica r is locked on $precommitQC$. Then, $(n - f)$ votes were cast for the matching $prepareQC$ in the PREPARE phase (Line 10 of Algorithm 2), out of which at least $f + 1$ were from correct replicas. □

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 $highQC$ before broadcasting a PREPARE messsage. Suppose among all replicas (including the leader itself), the highest kept lock is $lockedQC = precommitQC^*$. By Lemma 3, we know there are at least $f + 1$ correct replicas that voted for a $prepareQC^*$ matching $precommitQC^*$, and have already sent them to the leader in their NEW-VIEW messages. Thus, the leader must learn a matching $prepareQC^*$ in at least one of these NEW-VIEW messages and use it as $highQC$ in its PREPARE 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 PREPARE phase, since in SAFENODE, the condition on Line 27 of Algorithm 1 is satisfied (even if the $node$ in the message conflicts with a replica's stale $lockedQC.node$, and so Line 26 is not). Then, after the leader assembles a valid $prepareQC$ 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 is Optimistically Responsive because there is no explicit "wait-for-Δ" step, and the logical disjunction in SAFENODE is used to override a stale lock with the help of the three-phase paradigm. □

We now provide simple constructions for LEADER and 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 LEADER to return some deterministic mapping from view number to a replica, eventually rotating through all replicas. A possible solution for 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 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.

Livelessness 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.

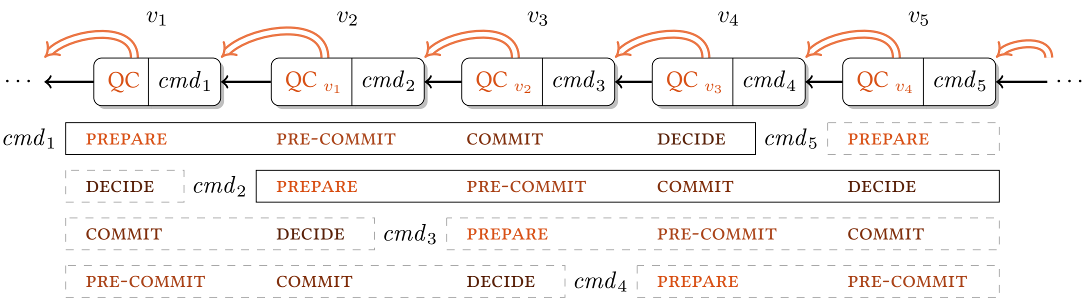
In the two-phase HotStuff variant, we omit the PRE-COMMIT phase and proceed directly to COMMIT. A replica becomes locked when it votes on a $prepareQC$. Suppose, in view v, a leader proposes b. It completes the PREPARE phase, and some replica r_v votes for the $prepareQC$, say qc, such that $qc.node = b$. Hence, r_v becomes locked on qc. An asynchronous network scheduling causes the rest of the replicas to move to view $v + 1$ without receiving qc.

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

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 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 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].



v_1
v_2
v_3
v_4
v_5
QC
cmd_1
QC v_1
cmd_2
QC v_2
cmd_3
QC v_3
cmd_4
QC v_4
cmd_5
cmd_1 PREPARE PRE-COMMIT COMMIT DECIDE
cmd_5 PREPARE
DECIDE cmd_2 PREPARE PRE-COMMIT COMMIT DECIDE
COMMIT DECIDE cmd_3 PREPARE PRE-COMMIT COMMIT
PRE-COMMIT COMMIT DECIDE cmd_4 PREPARE PRE-COMMIT

Figure 1: Chained HotStuff is a pipelined Basic HotStuff where a QC can serve in different phases simultaneously.

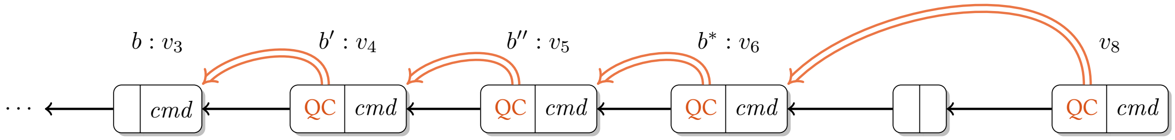


Figure 2: The nodes at views v_4, v_5, v_6 form a Three-Chain. The node at view v_8 does not make a valid One-Chain in Chained HotStuff (but it is a valid One-Chain after relaxation in the algorithm of Section 6).

More specifically, in Chained HotStuff the votes over a PREPARE phase are collected in a view by the leader into a $genericQC$. Then the $genericQC$ is relayed to the leader of the next view, essentially delegating responsibility for the next phase, which would have been PRE-COMMIT, to the next leader. However, the next leader does not actually carry a PRE-COMMIT phase, but instead initiates a new PREPARE phase and adds its own proposal. This PREPARE phase for view $v + 1$ simultaneously serves as the PRE-COMMIT phase for view v. The PREPARE phase for view $v + 2$ simultaneously serves as the PRE-COMMIT phase for view $v + 1$ and as the COMMIT phase for view 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 v_1, v_2, v_3 of Chained HotStuff serve as the PREPARE, PRE-COMMIT, and COMMIT Basic HotStuff phases for cmd_1 proposed in v_1. This command becomes committed by the end of v_4. Views v_2, v_3, v_4 serve as the three Basic HotStuff phases for cmd_2 proposed in v_2, and it becomes committed by the end of v_5. Additional proposals generated in these phases continue the pipeline similarly, and are denoted by dashed boxes. In Figure 1, a single arrow denotes the $b.parent$ field for a node b, and a double arrow denotes $b.justify.node$.

Hence, there are only two types of messages in Chained HotStuff, a NEW-VIEW message and generic-phase GENERIC message. The 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 COMMIT and DECIDE phases of Basic HotStuff.

Dummy nodes. The $genericQC$ used by a leader in some view $viewNumber$ may not directly reference the proposal of the preceding view $(viewNumber - 1)$. 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, CreateLeaf extends $genericQC.node$ 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 b may not refer to its parent, i.e., $b.justify.node$ may not equal $b.parent$ (the last node in Figure 2).

One-Chain, Two-Chain, and Three-Chain. When a node b^* carries a QC that refers to a direct parent, i.e., $b^*.justify.node = b^*.parent$, we say that it forms a *One-Chain*. Denote by $b'' = b^*.justify.node$. Node b^* forms a *Two-Chain*, if in addition to forming a One-Chain, $b''.justify.node = b''.parent$. It forms a *Three-Chain*, if b'' forms a Two-Chain.

Looking at chain $b = b'.justify.node$, $b' = b''.justify.node$, $b'' = b^*.justify.node$, 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 NextView.

If b^* forms a One-Chain, the PREPARE phase of b'' has succeeded. Hence, when a replica votes for b^*, it should remember $genericQC \leftarrow b^*.justify$. We remark that it is safe to update $genericQC$ even when a One-Chain is not direct, so long as it is higher than the current $genericQC$. In the implementation code described in Section 6, we indeed update $genericQC$ in this case.

If b^* forms a Two-Chain, then the PRE-COMMIT phase of b' has succeeded. The replica should therefore update $lockedQC \leftarrow b''.justify$. 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 b^* forms a Three-Chain, the COMMIT phase of b has succeeded, and b becomes a committed decision.

Algorithm 3 shows the pseudocode for Chained HotStuff. The proof of safety given by Theorem 5 in Appendix A 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.

Algorithm 3 Chained HotStuff protocol.

```
procedure CREATELEAF($parent$, $cmd$, $qc$)
    $b.parent \leftarrow$ branch extending with blanks from $parent$ to height $curView$
    $b.cmd \leftarrow cmd$
    $b.justify \leftarrow qc$
    return $b$

for $curView \leftarrow 1, 2, 3, \ldots$ do
  ▷ GENERIC phase
    as a leader   // $r =$ LEADER($curView$)
    // $M$ is the set of messages collected at the end of previous view by the leader of this view
        $highQC \leftarrow \left(\arg\max_{m \in M}\{m.justify.viewNumber\}\right).justify$
        if $highQC.viewNumber > genericQC.viewNumber$ then $genericQC \leftarrow highQC$
        $curProposal \leftarrow$ CREATELEAF($genericQC.node$, client's command, $genericQC$)
        // PREPARE phase
        broadcast MSG(GENERIC, $curProposal$, $\bot$)
    as a replica
        wait for message $m$ : MATCHINGMSG($m$, GENERIC, $curView$) from LEADER($curView$)
        $b^* \leftarrow m.node$; $b'' \leftarrow b^*.justify.node$; $b' \leftarrow b''.justify.node$; $b \leftarrow b'.justify.node$
        if SAFENODE($b^*$, $b^*.justify$) then
            send VOTEMSG(GENERIC, $b^*$, $\bot$) to LEADER($curView + 1$)
        // start PRE-COMMIT phase on $b^*$'s parent
        if $b^*.parent = b''$ then
            $genericQC \leftarrow b^*.justify$
        // start COMMIT phase on $b^*$'s grandparent
        if $(b^*.parent = b'') \land (b''.parent = b')$ then
            $lockedQC \leftarrow b''.justify$
        // start DECIDE phase on $b^*$'s great-grandparent
        if $(b^*.parent = b'') \land (b''.parent = b') \land (b'.parent = b)$ then
            execute new commands through $b$, respond to clients
    as the next leader
        wait for all messages: $M \leftarrow \{m \mid$ MATCHINGMSG($m$, GENERIC, $curView$)$\}$
            until there are $(n - f)$ votes: $V \leftarrow \{v \mid v.partialSig \neq \bot \land v \in M\}$
        $genericQC \leftarrow$ QC($V$)
  ▷ Finally
    NEXTVIEW interrupt: goto this line if NEXTVIEW($curView$) is called during "wait for" in any phase
    send MSG(GENERIC, $\bot$, $genericQC$) to LEADER($curView + 1$)
```

6 Implementation

HotStuff is a practical protocol for building efficient SMR systems. 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 $genericQC$ 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 $genericQC$ and $lockedQC$, 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_{lock}	locked node (similar to $lockedQC$).
b_{exec}	last executed node.
qc_{high}	highest known QC (similar to $genericQC$) kept by a Pacemaker.
b_{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, b_0 contains a hard-coded QC for itself, $b_{lock}, b_{exec}, b_{leaf}$ are all initialized to b_0, and qc_{high} contains the QC for 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 [25, 20, 15] is for replicas to increase the count of Δ’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-VIEW 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-VIEW message 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:     $b.parent \leftarrow parent$; $b.cmd \leftarrow cmd$;
 3:     $b.justify \leftarrow qc$; $b.height \leftarrow height$; return $b$
 4: procedure UPDATE($b^*$)
 5:     $b'' \leftarrow b^*.justify.node$; $b' \leftarrow b''.justify.node$
 6:     $b \leftarrow b'.justify.node$
        // 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$_v$(GENERIC, $b_{new}$, $\bot$))
17:     if $b_{new}.height > vheight \wedge (b_{new}$ extends $b_{lock} \vee$
18:       $b_{new}.justify.node.height > b_{lock}.height)$ then
19:         $vheight \leftarrow b_{new}.height$
20:         SEND(GETLEADER(), VOTEMSG$_u$(GENERIC, $b_{new}$, $\bot$))
21:     UPDATE($b_{new}$)
22: procedure ONRECEIVEVOTE($m$ = VOTEMSG$_v$(GENERIC, $b$, $\bot$))
23:     if $\exists\langle v, \sigma'\rangle \in V[b]$ then return  // avoid duplicates
24:     $V[b] \leftarrow V[b] \cup \{\langle v, m.partialSig\rangle\}$  // collect votes
25:     if $|V[b]| \geq n - f$ then
26:         $qc \leftarrow$ QC($\{\sigma \mid \langle v', \sigma\rangle \in V[b]\}$)
27:         UPDATEQCHIGH($qc$)
28: function ONPROPOSE($b_{leaf}$, $cmd$, $qc_{high}$)
29:     $b_{new} \leftarrow$ CREATELEAF($b_{leaf}$, $cmd$, $qc_{high}$, $b_{leaf}.height + 1$)
        // send to all replicas, including $u$ itself
30:     BROADCAST(MSG$_u$(GENERIC, $b_{new}$, $\bot$))
31:     return $b_{new}$
```

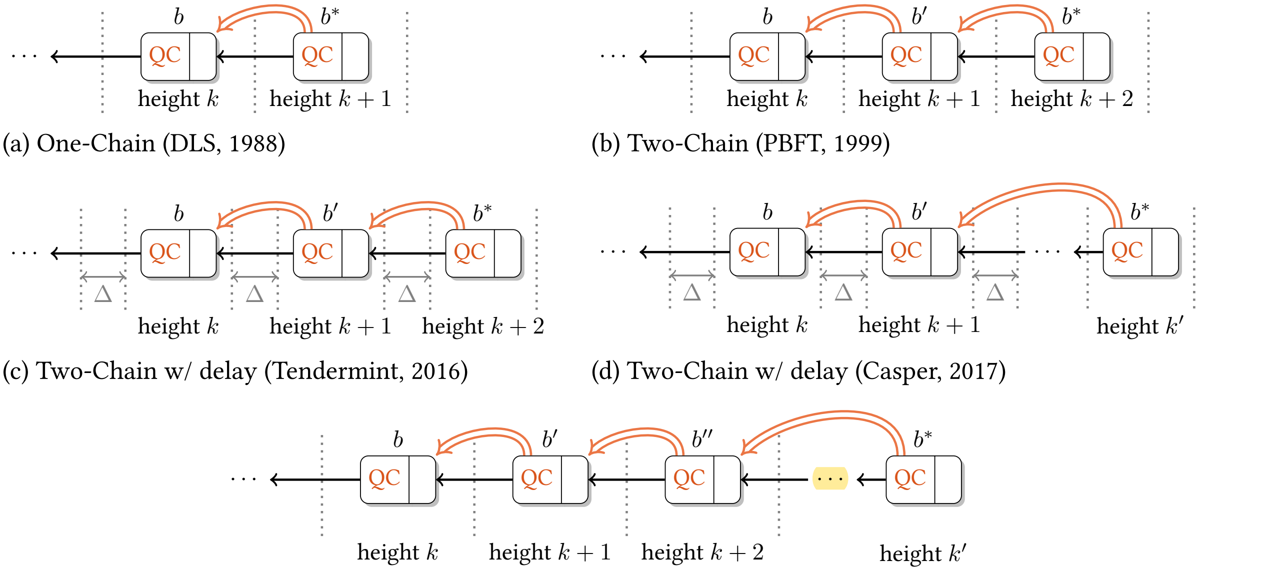
Algorithm 5 Code skeleton for a Pacemaker (for replica u).

```
// We assume Pacemaker in all correct replicas will have
synchronized leadership after GST.
function GETLEADER  // ...specified by the application
procedure UPDATEQCHIGH($qc'_{high}$)
    if $qc'_{high}.node.height > qc_{high}.node.height$ then
        $qc_{high} \leftarrow qc'_{high}$
        $b_{leaf} \leftarrow qc_{high}.node$
procedure ONBEAT($cmd$)
    if $u =$ GETLEADER() then
        $b_{leaf} \leftarrow$ ONPROPOSE($b_{leaf}, cmd, qc_{high}$)
procedure ONNEXTSYNCVIEW
    send MSG(NEW-VIEW, $\perp$, $qc_{high}$) to GETLEADER()
procedure ONRECEIVENEWVIEW(MSG(NEW-VIEW, $\perp$, $qc'_{high}$))
    UPDATEQCHIGH($qc'_{high}$)
```

Algorithm 6 UPDATE replacement for two-phase HotStuff.

```
procedure UPDATE($b^*$)
    $b' \leftarrow b^*.justify.node$ ; $b \leftarrow b'.justify.node$
    UPDATEQCHIGH($b^*.justify$)
    if $b'.height > b_{lock}.height$ then $b_{lock} \leftarrow b'$
    if ($b'.parent = b$) 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. See Section 7.3 for further discussion.



(a) One-Chain (DLS, 1988)

(b) Two-Chain (PBFT, 1999)

(c) Two-Chain w/ delay (Tendermint, 2016)

(d) Two-Chain w/ delay (Casper, 2017)

(e) Three-Chain (HotStuff, 2018)

Figure 3: Commit rules for different BFT protocols.

7 One-Chain and Two-Chain BFT Protocols

In this section, we examine four BFT replication protocols spanning four decades of research in Byzantine fault tolernace, casting them into a chained framework similar to Chained HotStuff.

Figure 3 provides a birds-eye view of the commit rules of five protocols we consider, including HotStuff.

In a nutshell, the commit rule in DLS [25] is One-Chain, allowing a node to be committed only by its own leader. The commit rules in PBFT [20], Tendermint [15, 16] and Casper [17] are almost identical, and consist of Two-Chains. They differ in the mechanisms they introduce for liveness, PBFT has leader "proofs" of quadratic size (no Linearity), Tendermint and Casper introduce a mandatory Δ delay before each leader proposal (no Optimistic Responsiveness). HotStuff uses a Three-Chain rule, and has a linear leader protocol without delay.

7.1 DLS

The simplest commit rule is a One-Chain. Modeled after Dwork, Lynch, and Stockmeyer (DLS), the first known asynchronous Byzantine Consensus solution, this rule is depicted in Figure 3(a). A replica becomes locked in DLS on the highest node it voted for.

Unfortunately, this rule would easily lead to a deadlock if at some height, a leader equivocates, and two correct replicas became locked on the conflicting proposals at that height. Relinquishing either lock is unsafe unless there are $2f + 1$ that indicate they did not vote for the locked value.

Indeed, in DLS only the leader of each height can itself reach a commit decision by the One-Chain commit rule. Thus, only the leader itself is harmed if it has equivocated. Replicas can relinquish a lock either if $2f + 1$ replicas did not vote for it, or if there are conflicting proposals (signed by the leader). The unlocking protocol occurring at the end of each height in DLS turns out to be fairly complex and expensive. Together with the fact that only the leader for a height can decide, in the best scenario where no fault occurs and the network is timely, DLS requires n leader rotations, and $O(n^4)$ message transmissions, per single decision. While it broke new ground in demonstrating a safe asynchronous protocol, DLS was not designed as a practical solution.

7.2 PBFT

Modeled after PBFT, a more practical appraoch uses a Two-Chain commit rule, see Figure 3(b). When a replica votes for a node that forms a One-Chain, it becomes locked on it. Conflicting One-Chains at the same height are simply not possible, as each has a QC, hence the deadlock situation of DLS is avoided.

However, if one replica holds a higher lock than others, a leader may not know about it even if it collects information from $n - f$ replicas. This could prevent leaders from reaching decisions ad infinitum, purely due to scheduling. To get "unstuck", the PBFT unlocks all replicas by carrying a *proof* consisting of the highest One-Chain's by $2f + 1$ replicas. This proof is quite involved, as explained below.

The original PBFT, which has been open-sourced [20] and adopted in several follow up works [13, 34], a leader proof contains a set of messages collected from $n - f$ replicas reporting the highest One-Chain each member voted for. Each One-Chain contains a QC, hence the total communication cost is $O(n^3)$. Harnessing signature combining methods from [45, 18], SBFT [30] reduces this cost to $O(n^2)$ by turning each QC to a single value.

In the PBFT variant in [21], a leader proof contains the highest One-Chain the leader collected from the quorum only once. It also includes one signed value from each member of the quorum, proving that it did not vote for a higher One-Chain. Broadcasting this proof incurs communication complexity $O(n^2)$. Note that whereas the signatures on a QC may be combined into a single value, the proof as a whole cannot be reduced to constant size because messages from different members of the quorum may have different values.

In both variants, a correct replica unlocks even it has a higher One-Chain than the leader's proof. Thus, a correct leader can force its proposal to be accepted during period of synchrony, and liveness is guaranteed. The cost is quadratic communication per leader replacement.

7.3 Tendermint and Casper

Tendermint has a Two-Chain commit rule identical to PBFT, and Casper has a Two-Chain rule in which the leaf does not need to have a QC to direct parent. That is, in Casper, Figure 3(c,d) depicts the commit rules for Tendermint and Casper, respectively.

In both methods, a leader simply sends the highest One-Chain it knows along with its proposal. A replica unlocks a One-Chain if it receives from the leader a higher one.

However, because correct replicas may not vote for a leader's node, to guarantee progress a new leader must obtain the highest One-Chain by waiting the maximal network delay. Otherwise, if leaders only wait for the first $n - f$ messages to start a new height, there is no progress guarantee. Leader delays are inherent both in Tendermint and in Casper, in order to provide liveness.

This simple leader protocol embodies a linear leap in the communication complexity of the leader protocol, which HotStuff borrows from. As already mentioned above, a QC could be captured in a single value using threshold-signatures, hence a leader can collect and disseminate the highest One-Chain with linear communication complexity. However, crucially, due to the extra QC step, HotStuff does not require the leader to wait the maximal network delay.

8 Evaluation

We have implemented HotStuff as a library in roughly 4K lines of C++ code. Most noticeably, the core consensus logic specified in the pseudocode consumes only around 200 lines. In this section, we will first examine baseline throughput and latency by comparing to a state-of-art system, BFT-SMaRt [13]. We then focus on the message cost for view changes to see our advantages in this scenario.

8.1 Setup

We conducted our experiments on Amazon EC2 using `c5.4xlarge` instances. Each instance had 16 vCPUs supported by Intel Xeon Platinum 8000 processors. All cores sustained a Turbo CPU clock speed up to 3.4GHz. We ran each replica on a single VM instance, and so BFT-SMaRt, which makes heavy use of threads, was allowed to utilize 16 cores per replica, as in their original evaluation [13]. The maximum TCP bandwidth measured by `iperf` was around 1.2 Gigabytes per second. We did not throttle the bandwidth in any run. The network latency between two machines was less than 1 ms.

Our prototype implementation of HotStuff uses secp256k1 for all digital signatures in both votes and quorum certificates. BFT-SMaRt uses hmac-sha1 for MACs (Message Authentication Codes) in the messages during normal operation and uses digital signatures in addition to MACs during a view change.

All results for HotStuff reflect end-to-end measurement from the clients. For BFT-SMaRt, we used the micro-benchmark programs `ThroughputLatencyServer` and `ThroughputLatencyClient` from the BFT-SMaRt website (https://github.com/bft-smart/library). The client program measures end-to-end latency but not throughput, while the server-side program measures both throughput and latency. We used the throughput results from servers and the latency results from clients.

8.2 Base Performance

We first measured throughput and latency in a setting commonly seen in the evaluation of other BFT replication systems. We ran 4 replicas in a configuration that tolerates a single failure, i.e., $f = 1$, while varying the operation request rate until the system saturated. This benchmark used empty (zero-sized) operation requests and responses and triggered no view changes; we expand to other settings below. Although our responsive HotStuff is three-phase, we also run its two-phase variant as an additional baseline, because the BFT-SMaRt baseline has only two phases.

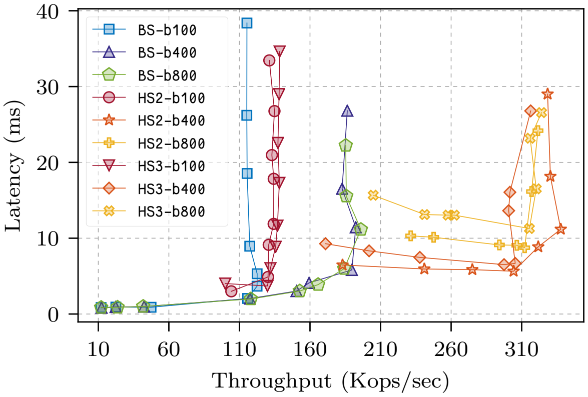


Figure 4: Throughput vs. latency with different choices of batch size, 4 replicas, 0/0 payload.

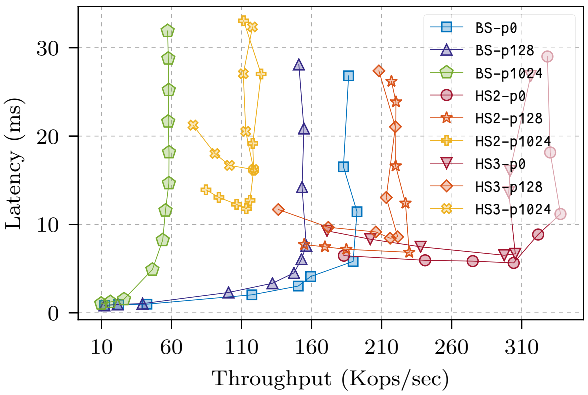


Figure 5: Throughput vs. latency with different choices of payload size, 4 replicas, batch size of 400.

Figure 4 depicts three batch sizes for both systems, 100, 400, and 800, though because these systems have different batching schemes, these numbers mean slightly different things for each system. BFT-SMaRt drives a separate consensus decision for each operation, and batches the messages from multiple consensus protocols. Therefore, it has a typical L-shaped latency/throughput performance curve. HotStuff batches multiple operations in each node, and in this way, mitigates the cost of digital signatures per decision. However, above 400 operations per batch, the latency incurred by batching becomes higher than the cost of the crypto. Despite these differences, both three-phase ("`HS3-`") and two-phase ("`HS2-`") HotStuff achieves comparable latency performance to BFT-SMaRt ("`BS-`") for all three batch sizes, while their maximum throughput noticeably outperformed BFT-SMaRt.

For batch sizes of 100 and 400, the lowest-latency HotStuff point provides latency and throughput that are better than the latency and throughput simultaneously achievable by BFT-SMaRT at its highest throughput, while incurring a small increase in latency. This increase is partly due to the batching strategy employed by HotStuff: It needs three additional full batches (two in the two-phase variant) to arrive at a decision on a batch. Our experiments kept the number of outstanding requests high, but the higher the batch size, the longer it takes to fill the batching pipeline. Practical deployments could be further optimized to adapt the batch size to the number of outstanding operations.

Figure 5 depicts three client request/reply payload sizes (in bytes) of 0/0, 128/128, and 1024/1024, denoted "p0", "p128", and "p1024" respectively. At all payload sizes, both three-phase and two-phase HotStuff outperformed BFT-SMaRt in throughput, with similar or comparable latency.

Notice BFT-SMaRt uses MACs based on symmetric crypto that is orders of magnitude faster than the asymmetric crypto in digital signatures used by HotStuff, and also three-phase HotStuff has more round trips compared to two-phase PBFT variant used by BFT-SMaRt. Yet HotStuff is still able to achieve comparable latency and much higher throughput. Below we evaluate both systems in more challenging situations, where the performance advantages of HotStuff will become more pronounced.

8.3 Scalability

To evaluate the scalability of HotStuff in various dimensions, we performed three experiments. For the baseline, we used zero-size request/response payloads while varying the number of replicas. The second evaluation repeated the baseline experiment with 128-byte and 1024-byte request/response payloads. The third test repeated the baseline (with empty payloads) while introducing network delays between replicas that were uniformly distributed in 5ms ± 0.5ms or in 10ms ± 1.0ms, implemented using `NetEm` (see https://www.linux.org/docs/man8/tc-netem.html). For each data point, we repeated five runs with the same setting and show error bars to indicate the standard deviation for all runs.

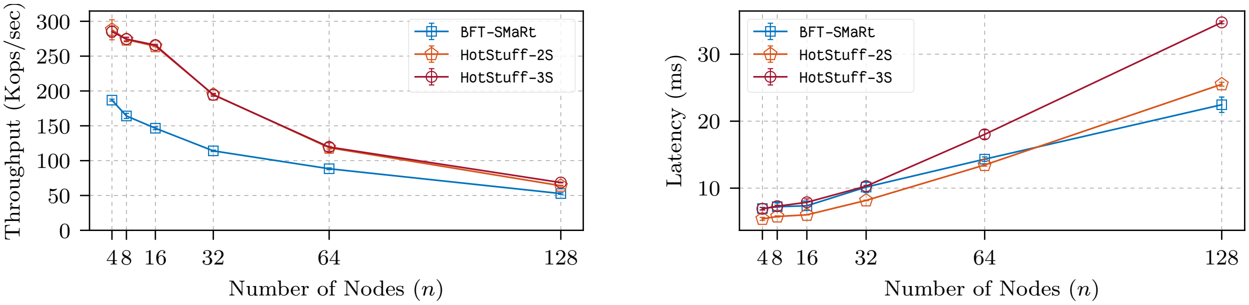
The first setting is depicted in Figure 6a (throughput) and Figure 6b (latency). Both three-phase and two-phase HotStuff show consistently better throughput than BFT-SMaRt, while their latencies are still comparable to BFT-SMaRt with graceful degradation. The performance scales better than BFT-SMaRt when $n < 32$. This is because we currently still use a list of secp256k1 signatures for a QC. In the future, we plan to reduce the cryptographic computation overhead in HotStuff by using a fast threshold signature scheme.

The second setting with payload size 128 or 1024 bytes is denoted by "`p128`" or "`p1024`" in Figure 7a (throughput) and Figure 7b (latency). Due to its quadratic bandwidth cost, the throughput of BFT-SMaRt scales worse than HotStuff for reasonably large (1024-byte) payload size.

The third setting is shown in Figure 8a (throughput) and Figure 8b (latency) as "`5ms`" or "`10ms`". Again, due to the larger use of communication in BFT-SMaRt, HotStuff consistently outperformed BFT-SMaRt in both cases.

8.4 View Change

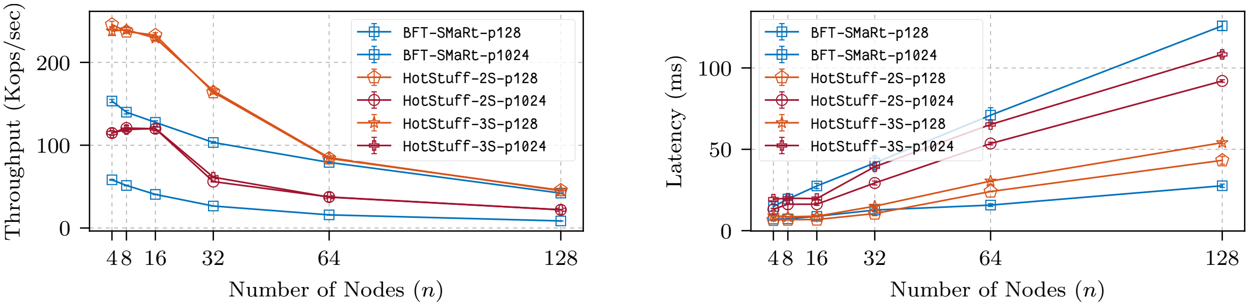
To evaluate the communication complexity of leader replacement, we counted the number of *MAC or signature verifications* performed within BFT-SMaRt's view-change protocol. Our evaluation strategy was as follows. We injected a view change into BFT-SMaRt every one thousand decisions. We instrumented the BFT-SMaRt source



(a) Throughput

(b) Latency

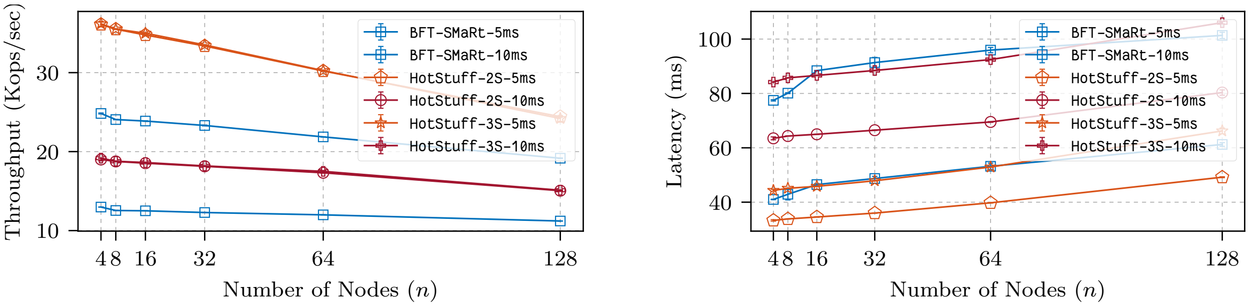
Figure 6: Scalability with 0/0 payload, batch size of 400.



(a) Throughput

(b) Latency

Figure 7: Scalability for 128/128 payload or 1024/1024 payload, with batch size of 400.



(a) Throughput

(b) Latency

Figure 8: Scalability for inter-replica latency 5ms ± 0.5ms or 10ms ± 1.0ms, with 0/0 payload, batch size of 400.

code to count the number of verifications upon receiving and processing messages within the view-change protocol. Beyond communication complexity, this measurement underscores the cryptographic computation load associated with transferring these authenticated values.

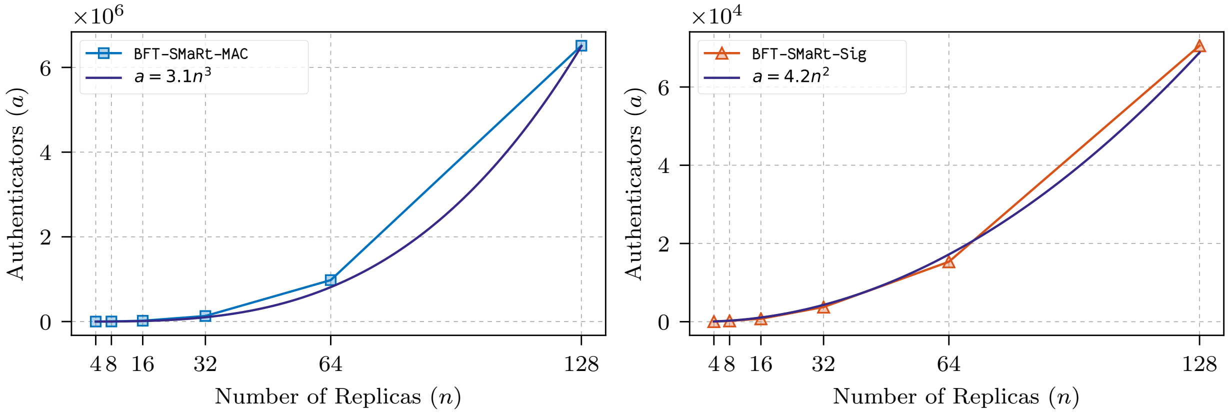
Figure 9a and Figure 9b show the number of extra authenticators (MACs and signatures, respectively) processed for each view change, where "extra" is defined to be those authenticators that would not be sent if the leader remained stable. Note that HotStuff has no "extra" authenticators by this definition, since the number of authenticators remains the same regardless of whether the leader stays the same or not. The two figures show that BFT-SMaRt uses cubic numbers of MACs and quadratic numbers of signatures. HotStuff does not require extra authenticators for view changes and so is omitted from the graph.

Evaluating the real-time performance of leader replacement is tricky. First, BFT-SMaRt got stuck when triggering frequent view changes; our authenticator-counting benchmark had to average over as many successful view changes as possible before the system got stuck, repeating the experiment many times. Second, the actual elapsed time for leader replacement depends highly on timeout parameters and the leader-election mechanism. It is therefore impossible to provide a meaningful comparison.

9 Conclusion

Since the introduction of PBFT, the first practical BFT replication solution in the partial synchrony model, numerous BFT solutions were built around its core two-phase paradigm. The first phase guarantees proposal uniqueness through a QC. The second phase guarantees that a new leader can convince replicas to vote for a safe proposal. This requires the leader to relay information from $(n-f)$ replicas, each reporting its own highest QC or vote. Generations of two-phase works thus suffer from a quadratic communication bottleneck on leader replacement.

HotStuff revolves around a three-phase core, allowing a new leader to simply pick the highest QC it knows of. This alleviates the above complexity and at the same time considerably simplifies the leader replacement protocol. Having (almost) canonized the phases, it is very easy to pipeline HotStuff, and to frequently rotate leaders.



(a) MACs

(b) Signatures

Figure 10: Number of extra authenticators used for each BFT-SMaRt view change.

Acknowledgements

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

[1] Casper ffg with one message type, and simpler fork choice rule. https://ethresear.ch/t/casper-ffg-with-one-message-type-and-simpler-fork-choice-rule/103, 2017.

[2] Istanbul bft's design cannot successfully tolerate fail-stop failures. https://github.com/jpmorganchase/quorum/issues/305, 2018.

[3] A livelock bug in the presence of byzantine validator. https://github.com/tendermint/tendermint/issues/1047, 2018.

[4] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Ramakrishna Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance. *CoRR*, abs/1712.01367, 2017.

[5] Ittai Abraham, Guy Gueta, Dahlia Malkhi, and Jean-Philippe Martin. Revisiting fast practical Byzantine fault tolerance: Thelma, velma, and zelma. *CoRR*, abs/1801.10022, 2018.

[6] Ittai Abraham and Dahlia Malkhi. The blockchain consensus layer and BFT. *Bulletin of the EATCS*, 123, 2017.

[7] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solida: A blockchain protocol based on reconfigurable byzantine consensus. In *21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal, December 18-20, 2017*, pages 25:1–25:19, 2017.

[8] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Validated asynchronous byzantine agreement with optimal resilience and asymptotically optimal time and word communication. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29-August 2, 2019*, 2019.

[9] Yair Amir, Brian A. Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE Trans. Dependable Sec. Comput.*, 8(4):564–577, 2011.

[10] Hagit Attiya, Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *J. ACM*, 41(1):122–152, 1994.

[11] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knezevic, Vivien Quéma, and Marko Vukolic. The next 700 BFT protocols. *ACM Trans. Comput. Syst.*, 32(4):12:1–12:45, 2015.

[12] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983*, pages 27–30, 1983.

[13] Alysson Neves Bessani, João Sousa, and Eduardo Adílio Pelinson Alchieri. State machine replication for the masses with BFT-SMART. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 355–362, 2014.

[14] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *J. Cryptology*, 17(4):297–319, 2004.

[15] Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, 2016.

[16] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018.

[17] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.

[18] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *J. Cryptology*, 18(3):219–246, 2005.

[19] Christian Cachin and Marko Vukolic. Blockchain consensus protocols in the wild. *CoRR*, abs/1707.01873, 2017.

[20] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 173–186, 1999.

[21] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[22] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Michael Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 277–290, 2009.

[23] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *J. ACM*, 32(1):191–204, 1985.

[24] Danny Dolev and H. Raymond Strong. Polynomial algorithms for multiple processor agreement. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 401–407, 1982.

[25] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[26] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, pages 45–59, 2016.

[27] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[28] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 281–310, 2015.

[29] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 51–68, 2017.

[30] Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: a scalable decentralized trust infrastructure for blockchains. *CoRR*, abs/1804.01626, 2018.

[31] Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY technology overview series, consensus system. *CoRR*, abs/1805.04548, 2018.

[32] Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. *J. Comput. Syst. Sci.*, 75(2):91–112, 2009.

[33] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. *CoRR*, abs/1602.06997, 2016.

[34] Ramakrishna Kotla, Lorenzo Alvisi, Michael Dahlin, Allen Clement, and Edmund L. Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4):7:1–7:39, 2009.

[35] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

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

[37] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[38] James Mickens. The saddest moment. *;login:*, 39(3), 2014.

[39] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 31–42, 2016.

[40] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

[41] Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II*, pages 643–673, 2017.

[42] Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, pages 3–33, 2018.

[43] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[44] HariGovind V. Ramasamy and Christian Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *Principles of Distributed Systems, 9th International Conference, OPODIS 2005, Pisa, Italy, December 12-14, 2005, Revised Selected Papers*, pages 88–102, 2005.

[45] Michael K. Reiter. The rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems, International Workshop, Dagstuhl Castle, Germany, September 5-9, 1994, Selected Papers*, pages 99–110, 1994.

[46] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In Bimal K. Roy and Willi Meier, editors, *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers*, volume 3017 of *Lecture Notes in Computer Science*, pages 371–388. Springer, 2004.

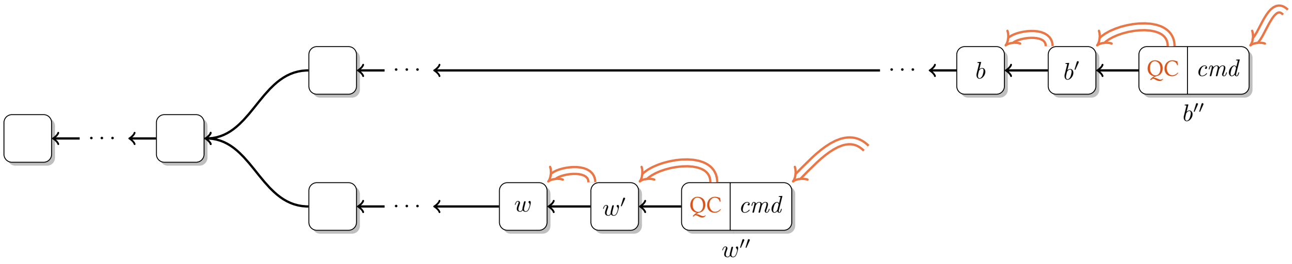


Figure 11: w and b both getting committed (impossible). Nodes horizontally arranged by view numbers.

[47] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[48] Victor Shoup. Practical threshold signatures. In *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, pages 207–220, 2000.

[49] Yee Jiun Song and Robbert van Renesse. Bosco: One-step byzantine asynchronous consensus. In *Distributed Computing, 22nd International Symposium, DISC 2008, Arcachon, France, September 22-24, 2008. Proceedings*, pages 438–450, 2008.

[50] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus in the lens of blockchain. *CoRR*, abs/1803.05069, 2018.

A Proof of Safety for Chained HotStuff

Theorem 5. *Let b and w be two conflicting nodes. Then they cannot both become committed, each by an honest replica.*

Proof. We prove this theorem by contradiction. By an argument similar to Lemma 1, b and w must be in different views. Assume during an exectuion b becomes committed at some honest replica via the QC Three-Chain b, b', b'', b^*, likewise, w becomes committed at some honest replica via the QC Three-Chain w, w', w'', w^*. Since each of b, b', b'', w, w', w'' get its QC, then w.l.o.g., we assume b is created in a view higher than w'', namely, $b'.justify.viewNumber > w^*.justify.viewNumber$, as shown in Figure 11.

We now denote by v_s the lowest view higher than $v_{w''} = w^*.justify.viewNumber$ in which there is a qc_s such that $qc_s.node$ conflicts with w. Let $v_b = b'.justify.viewNumber$. Formally, we define the following predicate for any qc:

$$E(qc) \coloneqq (v_{w''} < qc.viewNumber \leq v_b) \wedge (qc.node \text{ conflicts with } w).$$

We can now set the *first* switching point qc_s:

$$qc_s \coloneqq \arg\min_{qc}\{qc.viewNumber \mid qc \text{ is valid} \wedge E(qc)\}.$$

By assumption, such qc_s exists, for example, qc_s could be $b'.justify$. Let r denote a correct replica in the intersection of $w^*.justify$ and qc_s. By assumption on the minimality of qc_s, the lock that r has on w is not changed before qc_s is formed. Now consider the invocation of SAFENODE in view v_s by r, with a message m carrying a conflicting node $m.node = qc_s.node$. By assumption, the condition on the lock (Line 26 in Algorithm 1) is false. On the other hand, the protocol requires $t = m.node.justify.node$ to be an ancestor of $qc_s.node$. By minimality of qc_s, $m.node.justify.viewNumber \leq v_{w''}$. Since $qc_s.node$ conflicts with w, t cannot be w, w' or w''. Then, $m.node.justify.viewNumber < w'.justify.viewNumber$, so the other half of the disjunct is also false. Therefore, r will not vote for $qc_s.node$, contradicting the assumption of r.

□

The liveness argument is almost identical to Basic HotStuff, except that we have to assume after GST, two consecutive leaders are correct, to guarantee a decision. It is omitted for brevity.

B Proof of Safety for Implementation Pseudocode

Lemma 6. *Let b and w be two conflicting nodes such that $b.height = w.height$, then they cannot both have valid quorum certificates.*

Proof. Suppose they can, so both b and w receive $2f + 1$ votes, among which there are at least $f + 1$ honest replicas voting for each node, then there must be an honest replica that votes for both, which is impossible because b and w are of the same height. □

Notation 1. For any node b, let "$\leftarrow$" denote parent relation, i.e. $b.parent \leftarrow b$. Let "$\overset{*}{\leftarrow}$" denote ancestry, that is, the reflexive transitive closure of the parent relation. Then two nodes b, w are conflicting iff. $b \overset{*}{\not\leftarrow} w \wedge w \overset{*}{\not\leftarrow} b$. Let "$\Leftarrow$" denote the node a QC refers to, i.e. $b.justify.node \Leftarrow b$.

Lemma 7. *Let b and w be two conflicting nodes. Then they cannot both become committed, each by an honest replica.*

Proof. We prove this important lemma by contradiction. Let b and w be two conflicting nodes at different heights. Assume during an execution, b becomes committed at some honest replica via the QC Three-Chain $b(\Leftarrow \wedge \leftarrow)b'(\Leftarrow \wedge \leftarrow)b'' \Leftarrow b^*$; likewise, w becomes committed at some honest replica via the QC Three-Chain $w(\Leftarrow \wedge \leftarrow)w'(\Leftarrow \wedge \leftarrow)w'' \Leftarrow w^*$. By Lemma 1, since each of the nodes b, b', b'', w, w', w'' have QC's, then w.l.o.g., we assume $b.height > w''.height$, as shown in Figure 11.

We now denote by qc_s the QC for a node with the lowest height larger than $w''.height$, that conflicts with w. Formally, we define the following predicate for any qc:

$$E(qc) := (w''.height < qc.node.height \leq b.height) \wedge (qc.node \text{ conflicts with } w)$$

We can now set the first switching point qc_s:

$$qc_s := \underset{qc}{\arg\min}\{qc.node.height \mid qc \text{ is valid} \wedge E(qc)\}.$$

By assumption, such qc_s exists, for example, qc_s could be $b'.justify$. Let r denote a correct replica in the intersection of $w^*.justify$ and qc_s. By assumption of minimality of qc_s, the lock that r has on w is not changed before qc_s is formed. Now consider the invocation of onReceiveProposal, with a message carrying a conflicting node b_{new} such that $b_{new} = qc_s.node$. By assumption, the condition on the lock (Line 17 in Algorithm 4) is false. On the other hand, the protocol requires $t = b_{new}.justify.node$ to be an ancestor of b_{new}. By minimality of qc_s, $t.height \leq w''.height$. Since $qc_s.node$ conflicts with w, t cannot be w, w' or w''. Then, $t.height < w.height$, so the other half of the disjunct is also false. Therefore, r will not vote for b_{new}, contradicting the assumption of r. □

Theorem 8. *Let cmd_1 and cmd_2 be any two commands where cmd_1 is executed before cmd_2 by some honest replica, then any honest replica that executes cmd_2 must executes cmd_1 before cmd_2.*

Proof. Denote by w the node that carries cmd_1, b carries cmd_2. From Lemma 6, it is clear the committed nodes are at distinct heights. Without loss of generality, assume $w.height < b.height$. The commit of w are b are triggered by some onCommit(w') and onCommit(b') in update, where $w \overset{*}{\leftarrow} w'$ and $b \overset{*}{\leftarrow} b'$. According to Lemma 7, w' must not conflict with b', so w does not conflict with b. Then $w \overset{*}{\leftarrow} b$, and when any honest replica executes b, it must first executes w by the recursive logic in onCommit. □

B.1 Remarks

In order to shed insight on the tradeoffs taken in the HotStuff design, we explain why certain constraints are necessary for safety.

Why monotonic vheight? Suppose we change the voting rule such that a replica does not need to vote monotonically, as long as it does not vote more than once for each height. The weakened constraint will break safety. For example, a replica can first vote for b and then w. Before learning about b', b'', it first delivers w', w'', assuming the lock is on w, and vote for w''. When it eventually delivers b'', it will flip to the branch led by b because it is eligible for locking, and b is higher than w. Finally, the replica will also vote for b'', causing the commit of both w and b.

Why direct parent? The direct parent constraint is used to ensure the equality $b.height > w''.height$ used in the proof, with the help of Lemma 6. Suppose we do not enforce the rule for commit, so the commit constraint is weakened to $w \overset{*}{\leftarrow} w' \overset{*}{\leftarrow} w''$ instead of $w \leftarrow w' \leftarrow w''$ (same for b). Consider the case where $w'.height < b.height < b'.height < w''.height < b''.height$. Chances are, a replica can first vote for w'', and then discover b'' to switch to the branch by b, but it is too late since w could be committed.