HotStuff: BFT Consensus with Linearity and Responsiveness

Maofan Yin
Cornell University
VMware Research

Dahlia Malkhi
VMware Research

Michael K. Reiter
UNC-Chapel Hill
VMware Research

Guy Golan Gueta
VMware Research

Ittai Abraham
VMware Research

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

ACM Reference Format:
Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *2019 ACM Symposium on Principles of Distributed Computing (PODC '19), July 29-August 2, 2019, Toronto, ON, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3293611.3331591

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., [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], 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)

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs International 4.0 License.

PODC '19, July 29-August 2, 2019, Toronto, ON, Canada
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6217-7/19/07.
https://doi.org/10.1145/3293611.3331591

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 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.

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 $O(n^3)$ 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

Protocol	Authenticator complexity *Correct leader*	*Leader failure (view-change)*	*f leader failures*	Responsiveness
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.

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 [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 $F \subset [n]$ 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 [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 $\{\rho_i\}_{i \in I}$, where $|I| = k$ and each $\rho_i \leftarrow tsign_i(m)$, can be used to produce a digital signature $\sigma \leftarrow tcombine(m, \{\rho_i\}_{i \in I})$ on m. Any other replica can verify the signature using the public key and the function $tverify$. We require that if $\rho_i \leftarrow tsign_i(m)$ for each $i \in I$, $|I| = k$, and if $\sigma \leftarrow tcombine(m, \{\rho_i\}_{i \in I})$, then $tverify(m, \sigma)$ returns true. However, given oracle access to oracles $\{tsign_i(\cdot)\}_{i \in [n] \setminus F}$, an adversary who queries $tsign_i(m)$ 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 $tverify(m, \sigma)$ returns true). Throughout this paper, we use a threshold of $k = 2f + 1$. Again, we will typically leave invocations of $tverify$ 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 ← type
    m.viewNumber ← curView
    m.node ← node
    m.justify ← qc
    return m
function voteMsg(type, node, qc)
    m ← Msg(type, node, qc)
    m.partialSig ← tsign_r(⟨m.type, m.viewNumber, m.node⟩)
    return m
procedure createLeaf(parent, cmd)
    b.parent ← parent
    b.cmd ← cmd
    return b
function QC(V)
    qc.type ← m.type : m ∈ V
    qc.viewNumber ← m.viewNumber : m ∈ V
    qc.node ← m.node : m ∈ V
    qc.sig ← tcombine(⟨qc.type, qc.viewNumber, qc.node⟩,
                      {m.partialSig | m ∈ V})
    return qc
function matchingMsg(m, t, v)
    return (m.type = t) ∧ (m.viewNumber = v)
function matchingQC(qc, t, v)
    return (qc.type = t) ∧ (qc.viewNumber = v)
function safeNode(node, qc)
    return (node extends from lockedQC.node) ∨   // safety rule
    (qc.viewNumber > lockedQC.viewNumber)   // liveness rule
```

Algorithm 2 Basic HotStuff protocol (for replica r).

```
for curView ← 1, 2, 3, . . . 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 ← {m | matchingMsg(m, NEW-VIEW, curView−1)}
        highQC ← (arg max_{m∈M} {m.justify.viewNumber}).justify
        curProposal ← createLeaf(highQC.node,
                                 client's command)
        broadcast Msg(PREPARE, curProposal, highQC)
    as a replica
        wait for message m from leader(curView)
            m : matchingMsg(m, PREPARE, curView)
        if m.node extends from m.justify.node ∧
                safeNode(m.node, m.justify) then
            send voteMsg(PREPARE, m.node, ⊥) to leader(curView)

    ▷ PRE-COMMIT phase
    as a leader
        wait for (n − f) votes:
            V ← {v | matchingMsg(v, PREPARE, curView)}
        prepareQC ← QC(V)
        broadcast Msg(PRE-COMMIT, ⊥, prepareQC)
    as a replica
        wait for message m from leader(curView)
            m : matchingQC(m.justify, PREPARE, curView)
        prepareQC ← m.justify
        send to leader(curView)
            voteMsg(PRE-COMMIT, m.justify.node, ⊥)
    ▷ COMMIT phase
    as a leader
        wait for (n − f) votes:
            V ← {v | matchingMsg(v, PRE-COMMIT, curView)}
        precommitQC ← QC(V)
        broadcast Msg(COMMIT, ⊥, precommitQC)
    as a replica
        wait for message m from leader(curView)
            m : matchingQC(m.justify, PRE-COMMIT, curView)
        lockedQC ← m.justify
        send to leader(curView)
            voteMsg(COMMIT, m.justify.node, ⊥)
    ▷ DECIDE phase
    as a leader
        wait for (n − f) votes:
            V ← {v | matchingMsg(v, COMMIT, curView)}
        commitQC ← QC(V)
        broadcast Msg(DECIDE, ⊥, commitQC)
    as a replica
        wait for message m from leader(curView)
            m : matchingQC(m.justify, COMMIT, curView)
        execute new commands through m.justify.node,
        respond to clients
    ▷ Finally
    NEXTVIEW interrupt: goto this line if NEXTVIEW(curView) is
    called during "wait for" in any phase
    send Msg(NEW-VIEW, ⊥, 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:

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

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 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].

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 [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.

Algorithm 3 Chained HotStuff protocol.

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

for curView ← 1, 2, 3, . . . do
    ▷ GENERIC phase
    as a leader   // r = LEADER(curView)
        wait for (n − f) NEW-VIEW messages:
            M ← {m | MATCHINGMSG(m, NEW-VIEW, curView−1)}
        // M includes the previous leader NEW-VIEW message, if received
```

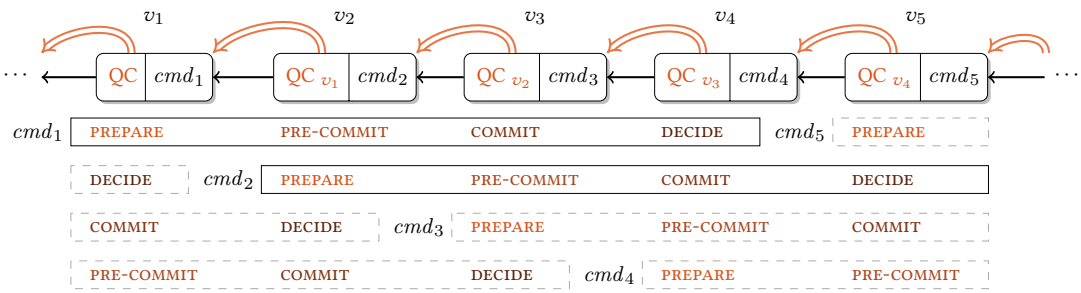

Figure 1: Chained HotStuff is a pipelined Basic HotStuff where a QC can serve in different phases simultaneously.

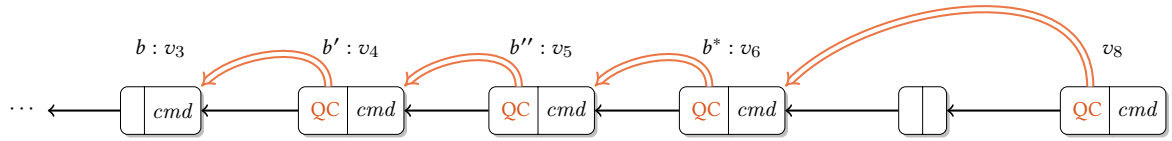

Figure 2: The nodes at views v_4, v_5, v_6 form a Three-Chain. The node at view v_8 does not make a valid One-Chain in Chained HotStuff (but it is a valid One-Chain after relaxation in the algorithm of Section 6).

```
        // while waiting in previous view
6:      genericQC ← (arg max_{m∈M} {m.justify.viewNumber}).justify
7:      curProposal ← CREATELEAF(genericQC.node,
                                 client's command, genericQC)
        // PREPARE phase (leader-half)
8:      broadcast MSG(GENERIC, curProposal, ⊥)
9:  as a replica
10:     wait for message m from LEADER(curView)
            m : MATCHINGMSG(m, GENERIC, curView)
11:     b* ← m.node; b'' ← b*.justify.node;
        b' ← b''.justify.node; b ← b'.justify.node
12:     if SAFENODE(b*, b*.justify) then
13:         send VOTEMSG(GENERIC, b*, ⊥) to LEADER(curView)
        // start PRE-COMMIT phase on b*'s parent
14:     if b*.parent = b'' then
15:         genericQC ← b*.justify
        // start COMMIT phase on b*'s grandparent
16:     if (b*.parent = b'') ∧ (b''.parent = b') then
17:         lockedQC ← b''.justify
        // start DECIDE phase on b*'s great-grandparent
18:     if (b*.parent = b'') ∧ (b''.parent = b') ∧
            (b'.parent = b) then
19:         execute new commands through b, respond to clients
20: as a leader   // PRE-COMMIT phase (leader-half)
21:     wait for (n − f) votes:
            V ← {v | MATCHINGMSG(v, GENERIC, curView)}
22:     genericQC ← QC(V)
    // for liveness, the message here counts as (n − f) at Line 5
23: as the next leader
24:     wait for message m from LEADER(curView)
            m : MATCHINGMSG(m, NEW-VIEW, curView)
  ▹ Finally
25: NEXTVIEW interrupt: goto this line if NEXTVIEW(curView) is
    called during "wait for" in any phase
26: send MSG(NEW-VIEW, ⊥, 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 [15, 20, 25] 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 ← parent; b.cmd ← cmd;
3:      b.justify ← qc; b.height ← height; return b
4:  procedure UPDATE(b*)
5:      b'' ← b*.justify.node; b' ← b''.justify.node
6:      b ← b'.justify.node
        // PRE-COMMIT phase on b''
7:      UPDATEQCHIGH(b*.justify)
8:      if b'.height > b_lock.height then
9:          b_lock ← b'  // COMMIT phase on b'
10:     if (b''.parent = b') ∧ (b'.parent = b) then
11:         ONCOMMIT(b)
12:         b_exec ← 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, ⊥))
17:     if b_new.height > vheight ∧ (b_new extends b_lock ∨
18:       b_new.justify.node.height > b_lock.height) then
19:         vheight ← b_new.height
20:         SEND(GETLEADER(), VOTEMSG_u(GENERIC, b_new, ⊥))
21:     UPDATE(b_new)
22: procedure ONRECEIVEVOTE(m = VOTEMSG_v(GENERIC, b, ⊥))
23:     if ∃⟨v, σ'⟩ ∈ V[b] then return   // avoid duplicates
24:     V[b] ← V[b] ∪ {⟨v, m.partialSig⟩}  // collect votes
25:     if |V[b]| ≥ n − f then
26:         qc ← QC({σ | ⟨v', σ⟩ ∈ V[b]})
27:         UPDATEQCHIGH(qc)
28: function ONPROPOSE(b_leaf, cmd, qc_high)
29:     b_new ← CREATELEAF(b_leaf, cmd, qc_high, b_leaf.height + 1)
        // send to all replicas, including u itself
30:     BROADCAST(MSG_u(GENERIC, b_new, ⊥))
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)
3:      if qc'_high.node.height > qc_high.node.height then
4:          qc_high ← qc'_high
5:          b_leaf ← qc_high.node
6:  procedure ONBEAT(cmd)
7:      if u = GETLEADER() then
8:          b_leaf ← ONPROPOSE(b_leaf, cmd, qc_high)
9:  procedure ONNEXTSYNCVIEW
10:     send MSG(NEW-VIEW, ⊥, qc_high) to GETLEADER()
11: procedure ONRECEIVENEWVIEW(MSG(NEW-VIEW, ⊥, qc'_high))
12:     UPDATEQCHIGH(qc'_high)
```

Algorithm 6 UPDATE replacement for two-phase HotStuff.

```
1:  procedure UPDATE(b*)
2:      b' ← b*.justify.node ; b ← b'.justify.node
3:      UPDATEQCHIGH(b*.justify)
4:      if b'.height > b_lock.height then b_lock ← b'
5:      if (b'.parent = b) then ONCOMMIT(b); b_exec ← 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 results to the longer paper [50]. There, we compare our implementation to BFT-SMaRt [13], 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

[1] 2017. 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] 2018. Istanbul BFT's Design Cannot Successfully Tolerate Fail-Stop Failures. https://github.com/jpmorganchase/quorum/issues/305. (2018).
[3] 2018. 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. 2017. Revisiting Fast Practical Byzantine Fault Tolerance. *CoRR* abs/1712.01367 (2017). arXiv:1712.01367
[5] Ittai Abraham, Guy Gueta, Dahlia Malkhi, and Jean-Philippe Martin. 2018. Revisiting Fast Practical Byzantine Fault Tolerance: Thelma, Velma, and Zelma. *CoRR* abs/1801.10022 (2018). arXiv:1801.10022

[6] Ittai Abraham and Dahlia Malkhi. 2017. The Blockchain Consensus Layer and BFT. *Bulletin of the EATCS* 123 (2017).

[7] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. 2017. 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*. 25:1–25:19. https://doi.org/10.4230/LIPIcs.OPODIS.2017.25

[8] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. 2019. 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*.

[9] Yair Amir, Brian A. Coan, Jonathan Kirsch, and John Lane. 2011. Prime: Byzantine Replication under Attack. *IEEE Trans. Dependable Sec. Comput.* 8, 4 (2011), 564–577. https://doi.org/10.1109/TDSC.2010.70

[10] Hagit Attiya, Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. 1994. Bounds on the Time to Reach Agreement in the Presence of Timing Uncertainty. *J. ACM* 41, 1 (1994), 122–152. https://doi.org/10.1145/174644.174649

[11] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knezevic, Vivien Quéma, and Marko Vukolic. 2015. The Next 700 BFT Protocols. *ACM Trans. Comput. Syst.* 32, 4 (2015), 12:1–12:45. https://doi.org/10.1145/2658994

[12] Michael Ben-Or. 1983. 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*. 27–30. https://doi.org/10.1145/800221.806707

[13] Alysson Neves Bessani, João Sousa, and Eduardo Adílio Pelinson Alchieri. 2014. 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*. 355–362. https://doi.org/10.1109/DSN.2014.43

[14] Dan Boneh, Ben Lynn, and Hovav Shacham. 2004. Short Signatures from the Weil Pairing. *J. Cryptology* 17, 4 (2004), 297–319. https://doi.org/10.1007/s00145-004-0314-9

[15] Ethan Buchman. 2016. *Tendermint: Byzantine fault tolerance in the age of blockchains*. Ph.D. Dissertation.

[16] Ethan Buchman, Jae Kwon, and Zarko Milosevic. 2018. The Latest Gossip on BFT Consensus. *CoRR* abs/1807.04938 (2018). arXiv:1807.04938

[17] Vitalik Buterin and Virgil Griffith. 2017. Casper the Friendly Finality Gadget. *CoRR* abs/1710.09437 (2017). arXiv:1710.09437

[18] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2005. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. *J. Cryptology* 18, 3 (2005), 219–246. https://doi.org/10.1007/s00145-005-0318-0

[19] Christian Cachin and Marko Vukolic. 2017. Blockchain Consensus Protocols in the Wild. *CoRR* abs/1707.01873 (2017). arXiv:1707.01873

[20] Miguel Castro and Barbara Liskov. 1999. 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*. 173–186. https://dl.acm.org/citation.cfm?id=296824

[21] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Trans. Comput. Syst.* 20, 4 (2002), 398–461. https://doi.org/10.1145/571637.571640

[22] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Michael Dahlin, and Taylor Riche. 2009. 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*. 277–290. https://doi.org/10.1145/1629575.1629602

[23] Danny Dolev and Rüdiger Reischuk. 1985. Bounds on Information Exchange for Byzantine Agreement. *J. ACM* 32, 1 (1985), 191–204. https://doi.org/10.1145/2455.214112

[24] Danny Dolev and H. Raymond Strong. 1982. 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*. 401–407. https://doi.org/10.1145/800070.802215

[25] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* 35, 2 (1988), 288–323. https://doi.org/10.1145/42282.42283

[26] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert van Renesse. 2016. 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*. 45–59.

[27] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (1985), 374–382. https://doi.org/10.1145/3149.214121

[28] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. 2015. 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*. 281–310. https://doi.org/10.1007/978-3-662-46803-6_10

[29] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. 51–68. https://doi.org/10.1145/3132747.3132757

[30] Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2018. SBFT: a Scalable Decentralized Trust Infrastructure for Blockchains. *CoRR* abs/1804.01626 (2018). arXiv:1804.01626

[31] Timo Hanke, Mahnush Movahedi, and Dominic Williams. 2018. DFINITY Technology Overview Series, Consensus System. *CoRR* abs/1805.04548 (2018). arXiv:1805.04548

[32] Jonathan Katz and Chiu-Yuen Koo. 2009. On Expected Constant-Round Protocols for Byzantine Agreement. *J. Comput. Syst. Sci.* 75, 2 (2009), 91–112. https://doi.org/10.1016/j.jcss.2008.08.001

[33] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. 2016. 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. 2009. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Trans. Comput. Syst.* 27, 4 (2009), 7:1–7:39. https://doi.org/10.1145/1658357.1658358

[35] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565. https://doi.org/10.1145/359545.359563

[36] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169. https://doi.org/10.1145/279227.279229

[37] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (1982), 382–401. https://doi.org/10.1145/357172.357176

[38] James Mickens. 2014. The Saddest Moment. *;login:* 39, 3 (2014). https://www.usenix.org/publications/login/june14/mickens

[39] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. 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*. 31–42. https://doi.org/10.1145/2976749.2978399

[40] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf. (2008).

[41] Rafael Pass, Lior Seeman, and Abhi Shelat. 2017. 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*. 643–673. https://doi.org/10.1007/978-3-319-56614-6_22

[42] Rafael Pass and Elaine Shi. 2018. 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*. 3–33. https://doi.org/10.1007/978-3-319-78375-8_1

[43] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. 1980. Reaching Agreement in the Presence of Faults. *J. ACM* 27, 2 (1980), 228–234. https://doi.org/10.1145/322186.322188

[44] HariGovind V. Ramasamy and Christian Cachin. 2005. 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*. 88–102. https://doi.org/10.1007/11795490_9

[45] Michael K. Reiter. 1994. 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*. 99–110. https://doi.org/10.1007/3-540-60042-6_7

[46] Phillip Rogaway and Thomas Shrimpton. 2004. Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. In *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers (Lecture Notes in Computer Science)*, Bimal K. Roy and Willi Meier (Eds.), Vol. 3017. Springer, 371–388. https://doi.org/10.1007/978-3-540-25937-4_24

[47] Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 22, 4 (1990), 299–319. https://doi.org/10.1145/98163.98167

[48] Victor Shoup. 2000. 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*. 207–220. https://doi.org/10.1007/3-540-45539-6_15

[49] Yee Jiun Song and Robbert van Renesse. 2008. Bosco: One-Step Byzantine Asynchronous Consensus. In *Distributed Computing, 22nd International Symposium, DISC 2008, Arcachon, France, September 22-24, 2008. Proceedings*. 438–450. https://doi.org/10.1007/978-3-540-87779-0_30

[50] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2018. HotStuff: BFT Consensus in the Lens of Blockchain. *CoRR* abs/1803.05069 (2018). arXiv:1803.05069