HotStuff-2: Optimal Two-Phase Responsive BFT

Dahlia Malkhi
Chainlink Labs

Kartik Nayak
Duke University

ABSTRACT

In this paper, we observe that it is possible to solve partially-synchronous BFT and simultaneously achieves $O(n^2)$ worst-case communication, optimistically linear communication, a two-phase commit regime within a view, and optimistic responsiveness. Prior work falls short in achieving one or more of these properties, e.g., the most closely related work, HotStuff, requires a three-phase view while achieving all other properties. We demonstrate that these properties are achievable through a two-phase HotStuff variant named HotStuff-2.

The quest for two-phase HotStuff variants that achieve all the above desirable properties has been long, producing a series of results that are yet sub-optimal and, at the same time, are based on somewhat heavy hammers. HotStuff-2 demonstrates that none of these are necessary: HotStuff-2 is remarkably simple, adding no substantive complexity to the original HotStuff protocol.

The main takeaway is that two phases are enough for BFT after all.

1 INTRODUCTION

In this work, we observe that it is possible to solve the known log replication problem in the partial synchrony model, simultaneously achieving the following measures to reach a single commit decision: An optimal $O(n^2)$ communication cost and a latency of $O(n)$ against a worst-case cascade of failures, optimistically linear communication, and a substantial improvement in optimistic latency (two phases) over previously known art [9, 18, 23]. At the same time, failure-free leader rotation should incur no extra delay, so that forming sequences of commit decisions additionally exhibit load balance at no extra complexity, each participant suffering the same communication load over time. We demonstrate that these properties are achievable through a two-phase HotStuff variant named HotStuff-2.

Existing protocols obtain various subsets of these features, but to our knowledge, no previous protocol achieves all of them simultaneously. In fact, the folklore belief is that this would be impossible (more on this later). Of particular relevance is HotStuff [23], which HotStuff-2 improves on. HotStuff satisfies all of the above features except the two-phase optimistic latency. In a nutshell, HotStuff-2 improves the latency of the HotStuff-based family of protocols by 33% at no extra complexity.

It is important to note that the complexity measures indicated above account for the full complexity incurred by the protocol. In particular, like most protocols in the partial synchrony setting, the protocol operates in a view-by-view manner, each view having a leader driving progress towards a commit decision. A key ingredient of solutions is a sub-protocol for view-synchronization (a "pacemaker"). Parties coordinate via the pacemaker to enter the view roughly at the same time as the leader. We present a full solution which does not punt on pacemaker details.

1.1 Relationship to Closely Related Results

The importance of the contribution is not in a new algorithmic technique but rather in helping demystify the belief that there may not exist a solution satisfying the following features simultaneously:

F-0 two-phase view regime
F-1 optimistically no wait over sequences of decisions
F-2 optimistically linear communication
F-3 balanced communication load over sequences of decisions
F-4 $O(n^2)$ worst-case communication

Indeed, HotStuff-2 satisfies all of these features. It is worthy of pointing to the evolution of the myth that led to believe this would be impossible.

- PBFT [7], the golden standard in BFT introduced two decades ago, emphasizes steady-state performance and obtains F-0 and F-1.
- Several enhancements on PBFT including FaB [19], Zyzzyva [17], SBFT [15], add support on top of F-0, F-1, for F-2 as well.
- Tendermint [4], Casper [5] and the original 2-phase version of HotStuff [1] enable F-3 through a simpler view-change regime. However, each new view has to wait for the maximal network delay Δ, even under a planned leader rotation. Hence, they do not satisfy F-1, namely, each view-change requires an explicit timer delay for the maximal network latency.
  It is worth remarking that these protocols do not achieve F-4 per se. First, as noted in [23], Tendermint/Casper would need to incorporate signature aggregation to support F-2. Second, an optimal pacemaker protocol is needed to support F-4.

HotStuff removes the need for each view-change to delay, thereby satisfying F-1 in addition to F-2 and F-3. RareSync [9] and Lewis-Pye [18] demonstrate a pacemaker for HotStuff that satisfies F-4 as well. However, the HotStuff family of protocols adds one phase to the view regime, hence giving up F-0.

There has been a long line of HotStuff variants aiming to improve HotStuff's view regime to two phases.

- Fast HotStuff [16], DiemBFT-v4 [22], and Jolteon and Ditto [13], provide a two-phase regime satisfying F-0 but revert to a PBFT quadratic view-change (Ditto also adding resilience against asynchrony). Hence, they trade F-4, namely they incur $O(n^2)$ communication every time a leader is faulty. A fortiori, an unlucky cascade of faulty leaders incurs $O(n^3)$ communication.
- Wendy [14] and MSCFCL [2] also revert to a PBFT view-change with a leader proof to convince parties of a safe proposal, but focus on compressing the leader proof. These schemes employ somewhat heavy hammers: Wendy introduces a novel signature scheme that works only when the gap between views that make progress is constant bounded

Protocol	View-change complexity	Responsive view-change	Phases
PBFT [7]	$O(n^2)$	✓	2
Tendermint [4]	$O(n)$	✗	2
HotStuff [23]	$O(n)$	✓	3
HotStuff-2 (this work)	$O(n)$	✓	2

Table 1: Comparison with closely related works. Note that PBFT and Tendermint have a view-change complexity of $O(n^3)$ and $O(n^2)$ respectively; however, both protocols can be easily improved by a factor of $O(n)$ to obtain the bounds described above.

and MSCFCL utilizes succinct arguments of knowledge whose complexity blows up quickly.

All of these advances are much more complex than HotStuff-2, whose key takeaway is that none of them is necessary. Indeed, a virtue of HotStuff-2 is simplicity, adding only a handful of statements to a production system currently being developed at Espresso Systems [21] as their core consensus algorithm.

2 INTUITION

In this section, we describe the key intuition behind our result. For simplicity, we will present it as a single-shot protocol.

At the core, view-by-view consensus works as follows. A view consists of two abstract steps. In the first step, a designated *leader* attempts to *reconcile* an output value, and in the second step, parties check whether there is an agreed value and *ratify* it. A core ingredient of the second step is *commit-adopt* [12], namely, if any honest party commits an output, $2t+1$ parties adopt and protect it; we say that parties *lock* the value. Likewise, a core ingredient of the first step is a *validity* condition, such that a leader's reconciled value is accepted subject to safety with respect to commit values of previous views and to non-equivocation. A full consensus proceeds by alternating the two steps: A commit-adopt value is carried into a reconciliation step of the next view in order to protect a commit decision; A reconciliation value is carried into a commit-adopt step to drive progress.

Different protocols differ in how commit-adopt is implemented in the second step, and how a leader generates and drives accepance of a valid proposal in the first.

PBFT [7]. In PBFT, the leader collects a status of locks from $2t+1$ parties at the start of a view. Observe that if some party has committed in earlier views, among the $2t+1$ locks in the status, up to t can be malicious, up to t can be from honest parties that are not guarding the safety of the commit, but there is at least one that corresponds to the committed value. On the other hand, if no party had committed in an earlier view, even if some parties locked on to the value, their status may or may not be a part of the locks the leader receives. In the former case, the leader proposes the highest lock it obtains. In the latter case, the leader is guaranteed that no honest party must have committed the value (based on the argument above) and is free to propose any value.

How does the leader convince the honest parties to vote for the value it proposed? It simply sends a "status certificate" containing the $2t+1$ locks in its proposal. Observe that every honest party can apply the same reasoning as the leader did, and thus can vote for the value. In particular, this holds true even in situations when a party is locked on a value and the leader proposes a different value that is justified by the status certificate. In other words, the status certificate from an honest leader provides sufficient information to all honest parties to vote on the proposal.

Note that sending this message can be responsive since the leader can act as soon as it receives $2t+1$ locks. However, the status certificate has linear complexity and thus sending it in the proposal leads to quadratic communication complexity.

Tendermint [4]. Tendermint improves PBFT by not requiring the leader to send a $(2t+1)$-sized status certificate and thus can perform a view change with linear complexity. Well, how does a leader know what to propose, and how can it be sure that all honest parties would indeed vote for it? Observe that by receiving $2t+1$ locks, an honest leader perhaps knows what to propose, but if parties do not have access to those locks (through a status certificate), they may not vote for it. In particular, in a situation where some honest party p is locked on a value, and a leader proposes a conflicting value, p cannot distinguish whether it needs to guard the safety of the value or go ahead and vote for the proposal. A generalization of this dilemma for this party is presented as a livelessness attack in HotStuff [23].

To address this concern, Tendermint requires the leader to wait for an $O(\Delta)$ time at the start of the view. After GST, the $O(\Delta)$ wait ensures that the leader receives locks from *all* honest parties. Thus, it can send a proposal that conforms with the value corresponding to the lock from the highest view, and send this lock together with the proposal. The highest-ranked lock convinces all honest parties to vote on the proposal sent by the leader. Note that, in this solution, while the leader learns the globally highest-ranked lock, the parties do not. However, the amount of information is sufficient to ascertain that the proposal is safe to vote on.

Tendermint obtains a linear communication complexity for view change compared to PBFT's quadratic communication complexity. However, due to the $O(\Delta)$ wait, the protocol is not responsive.

HotStuff [23]. The HotStuff protocol simultaneously obtains a linear view change and responsiveness when the leader is honest after GST. HotStuff addresses the livelessness concern differently while still ensuring that the value corresponding to the highest lock is proposed by the leader. Abstractly speaking, it uses the same argument as the one used for safety. For safety, we said, "if some party commits, then $\geq 2t+1$ parties hold a lock that will guard the safety of the commit, and ensure that the next leader receives it.". For liveness, HotStuff introduces another phase of votes and obtains a similar invariant: "if some party locks, then $\geq 2t+1$ parties know about the existence of this lock and thus hold a *key* corresponding to it. This key will be shared with the next leader to decide on a proposal appropriately." Correspondingly, the next leader would learn about the highest lock through the $2t + 1$ keys it receives, and the proposal would respect the globally highest lock held by any honest party. Note that, while locks guard the safety of a commit, keys do not, and thus honest parties only holding a higher key on a different value than the proposal can still vote for the proposal.

Thus, HotStuff obtains linear view change and responsiveness, but introducing the "key phase" makes it a three-phase protocol instead.

Our solution: HotStuff-2. Our work takes a fresh look at the livelessness concern and asks, *do we really need to add another phase to address this concern while still obtaining linear communication complexity and optimistic responsiveness?* Following the above discussion, if a leader knows about the highest lock *and* it can convince all honest parties about it, then the problem is solved. Indeed if a leader would receive a lock corresponding to the previous view, there *cannot* exist an even higher lock. Thus, a proposal respecting this lock would be voted for by all honest parties and the livelessness concern does not exist. If the leader does not obtain such a lock from the previous view, it would wait to hear about the locks from all honest parties by waiting $O(\Delta)$ time — this can happen when the leader from the previous view is malicious or the previous view was before GST. However, in those cases, we cannot hope to obtain responsiveness anyway. Thus, if we have a sequence of honest leaders after GST, each of them is guaranteed to drive progress responsively and generate a certificate in a view that will aid the next leader. Thus, all leaders except the first one in the chain can make honest parties commit responsively.

Our key observation is subtle, but once the insight is understood, really simple (at least we think so!), allowing us to solve the problem without any heavy machinery.

3 MODEL AND PERFORMANCE MEASURES

Briefly, in log replication, a group of $n = 3t + 1$ parties, t of which are Byzantine (arbitrarily faulty), reach agreement on a growing sequence of values referred to as "blocks". We refer to the non-faulty parties as *correct* or *honest*.

We operate in the partial synchrony model [11] where there is a known bound Δ on message transmission delays, such that after an unknown time called "GST" all transmissions arrive within Δ bound to their destinations. Parties output increasing log prefixes with the following guarantees:

- **Safety.** At all times, the outputs of every pair of correct parties is a prefix of one another.
- **Liveness.** After GST, agreement decisions to extend the log one position are repeatedly delivered by at least one correct party.

Throughout, we use the notation $\langle x \rangle_P$ to denote a signature or a threshold signature share on message x by party P. Wherever it is clear from context, we omit the signer P.

Performance can be measured after GST, since no progress is guaranteed until then. We are interested in preserving all the good performance features of HotStuff as described below, while improving on a single front – its optimistic latency.

Optimistically optimal performance per single decision. When a sequence of leaders are correct, i.e., in the happy-path protocol, HotStuff reaches a decision with linear communication and latency that depends only on actual network delays during execution (i.e., without ever waiting for the maximal network delay Δ).

Balanced per-party communication load in a sequence of decisions. Uniquely, HotStuff's happy-path linear regime and network speed properties can extend to a sequence of consensus decisions, while balancing communication load among parties. A sequence of $s = \Omega(n)$ decisions with honest leaders incurs overall $O(s \cdot n)$ communication, while each party suffers $O(s)$ communication load.

Optimal worst-case performance per single decision. When there are Byzantine leaders, there is an unavoidable performance cost in both communication complexity and latency. When faced with an unlucky cascade of t failures, $\Omega(n^2)$ communication cost is mandated by the Dolev-Reischuk lower bound [10], and $\Omega(n\Delta)$ latency is mandated by the Aguilera-Toueg bound [3]. Before HotStuff, the golden standard established by PBFT for handling a steady-leader replacement incurred $O(n^2)$ communication, hence a cascade of t failures would suffer $O(n^3)$ communication cost. Leveraging the HotStuff linear leader-replacement regime, two recent work, RareSync [9] and Lewis-Pye [18] closed this gap, achieving worst-case $O(n^2)$ communication and $O(n\Delta)$ latency.

Worst-case performance on a sequence of decisions. Furthermore, HotStuff allows to amortize the $O(n^2)$ communication cost over $O(n)$ decisions.

Remark – On failure cascades. It's worth noting that many protocols randomize the election of next leaders in order to reduce the chance of such a cascade. This improves the expected complexity in all protocols, including HotStuff, but not the worst-case. To make an apples-to-apples comparison, worst-case complexities are measured against t actual leader failures.

Optimistic latency. In many practical settings, low latency happy-path performance is crucial. Unfortunately, in HotStuff, this incurs 3-phases, each consisting of two communication rounds. In contrast, PBFT is two-phased, but incurs higher communication. Tendermint incurs two phases, has linear communication, but is not responsive. So this presents a trade-off between communication optimality and latency.

4 THE HOTSTUFF-2 PROTOCOL

Our solution operates in a view-by-view manner. It consists of two parts, a steady-state protocol and a pacemaker protocol for advancing views. That is, we present a full solution that does not punt on pacemaker details. Hence, the performance analysis we present portrays the full complexity incurred by the protocol. The steady-state protocol drives a commit decision in a view when all correct parties overlap in the view for sufficiently long, and a designated leader L_v for view v known to all parties is correct. The steady-state leader protocol is *almost* identical to a vanilla two-phase HotStuff protocol (where we simply take HotStuff and use a 2-chain commit rule instead of 3-chain). For ease of exposition, we let the leader drive the commit of an entire block in a view in two phases; we can easily modify this protocol to obtain a pipelined version akin to pipelined HotStuff protocol [23].

We will first describe some data structures and terminologies.

Block format. The protocol forms a chain of values. We use the term *block* to refer to each value in the chain. We refer to a block's position in the chain as its *height*. A block B_k at height k has the following format

$$B_k := (b_k, h_{k-1})$$

where b_k denotes a proposed value at height k and $h_{k-1} := H(B_{k-1})$ is a hash digest of the predecessor block. The first block $B_1 = (b_1, \perp)$ has no predecessor. Every subsequent block B_k must specify a predecessor block B_{k-1} by including a hash of it. We say a block is *valid* if (i) its predecessor is valid or $\perp$, and (ii) its proposed value meets application-level validity conditions and is consistent with its chain of ancestors (e.g., does not double-spend a transaction in one of its ancestor blocks).

Block extension and equivocation. We say B_l *extends* B_k, if B_k is an ancestor of B_l ($l > k$). We say two blocks B_l and $B'_{l'}$ *equivocate* one another if they are not equal and do not extend one another.

Certificates and certified blocks. In the protocol, parties vote for blocks by signing them using a threshold signature. We use $C_v(B_k)$ to denote a set of signatures on $h_k = H(B_k)$ by $2t + 1$ parties in view v. We call $C_v(B_k)$ a certificate for B_k from view v. Certified blocks are ranked by the views in which they are certified, i.e., a certificate $C_v(B_k)$ is ranked higher than $C_{v'}(B'_{k'})$ if $v > v'$.

Timeout certificates. In the protocol, parties may timeout if there is no progress in a particular view and use a pacemaker to justify and coordinate entrance to the next view. We use TC_v to denote a certificate by $2t + 1$ parties about wishing to enter view v.

Locked blocks. At any time, a party locks the highest certified block to its knowledge. During the protocol execution, each party keeps track of all signatures for all blocks and keeps updating its locked block. Looking ahead, the notion of a locked block will be used to guard the safety of a commit.

Steady-state protocol. Each view has a designated leader who is responsible for driving consensus on a sequence of blocks. Leaders can be chosen statically, e.g., round-robin, or randomly using more sophisticated techniques [6, 8]. In our description, we assume a round-robin selection of leaders for simplicity, i.e., (v mod n) is the leader of view v.

The goal of a view leader is to extend the highest certified block in the current sequence with a new block and get it certified by parties.

- A leader in view v forms a block B_k at height k that contains its proposed value, the view number, and the highest-ranked certificate the leader knows. The leader sends a proposed block to all the parties.
- Each party keeps the highest certificate it ever received. A party votes for block B_k if it extends the highest certificate, by sending a signed vote to the leader.
- A block becomes certified if $2t + 1$ parties vote for it. The certificate $C_v(B_k)$ formed from $2t + 1$ signed votes is aggregated by the leader.
- The leader engages in another round of votes; however, parties send these votes to the leader of view $v + 1$.

Pacemaker. The pacemaker is responsible for synchronizing entrance to views. Advancing to the next view $v + 1$ happens in one of two ways.

- **Case 1.** If $C_v(C_v(B_k))$ is obtained, the pacemaker can advance to the next view immediately. This guarantees optimistically optimal performance when there are no failures and the network is synchronous. Note that because the certificate is broadcast by the leader, the pacemaker does not require any further action to synchronize entering to the next view.
- **Case 2.** If such a certificate is not obtained, the pacemaker must wait at least a delay $\tau = \Omega(\Delta)$ time to allow a correct leader to make progress in a view under synchronous conditions. It is important to observe that incurring an $O(\Delta)$ delay upon a leader's failure is inevitable to allow correct leaders time to drive progress; otherwise, liveness would be compromised. It is also mandated by an $\Omega(n\Delta)$ latency lower bound Aguilera and Toueg [3].
  When τ-timers expire at a quorum of parties, the pacemaker can advance to the next view. To do this, the pacemaker needs to coordinate parties, so they enter the new view at roughly the same time.

There are various protocols for view-synchronization, including a trivial one that synchronizes to a wall clock but note this would not be optimistically responsive. The description below borrows the view-synchronization approach of RareSync and Lewis-Pye [9, 18], though other schemes could be used. Their approach is remarkably simple and elegant. It bundles consecutive views into epochs, where each epoch consists of $t + 1$ consecutive views. Parties employ a Cogsworth-like [20] coordination protocol in the first view of each epoch, and then they advance through the rest of the views in the same epoch using timeouts if there is no progress in the underlying consensus protocol. The worst-case message complexity is $O(n^2)$ messages per agreement decision, with $O(n\Delta)$ latency. Below, we integrate the view-synchronization details into the pacemaker module for completeness.

We now get to the HotStuff-2 mechanism that guarantees that a correct leader can form a certified block in a view after GST. We weave our mechanism into the pacemaker, such that it provides the new leader with a certified block for chaining at a linear communication cost. This works as follows. In Case 1 above, the new leader already obtains the certified block. In Case 2, advancing a view without forming a certificate already incurs a mandatory latency of $\tau = \Omega(\Delta)$. Our key observation is that a leader can recognize that it is in the Case 2 scenario. Since all parties enter the view within Δ time of the leader, and each party sends the leader its status, upon entering the new view with a TC, the leader can wait 2Δ time for all

Let v be the current view number and replica L_v be the leader in this view. Perform the following steps.

(1) **Enter.** Upon entering view v:
- **Leader L_v.** If entering view v using a TC or pacemaker timer expiration, the leader sets a timer 2Δ, and then proceeds to the propose step. Otherwise, it proceeds directly to the propose step.
- **Party.** If entering view v using a TC, the party sends its locked certificate to the leader L_v and proceeds to the vote step.

(2) **Propose.** The leader L_v broadcasts $\langle \text{propose}, B_k, v, C_{v'}(B_{k-1}), C_{v''}(C_{v''}(B_{k''}))\rangle_{L_v}$. ▷ Executed by the leader
Here, $B_k := (b_k, h_{k-1})$ is the block that should extend the highest certified block B_{k-1} with certificate $C_{v'}(B_{k-1})$ known to leader and $C_{v''}(C_{v''}(B_{k''}))$ is the largest double certificate known to the leader.

(3) **Vote and commit.** Upon receiving the first valid proposal $\langle \text{propose}, B_k, v, C_{v'}(B_{k-1})\rangle_{L_v}$ in view v: ▷ Executed by all parties
- If $C_{v'}(B_{k-1})$ is ranked no lower than the locked block, then send $\langle \text{vote}, B_k, v\rangle$ as a threshold signature share to L_v. Update lock to B_k and the certificate to $C_{v'}(B_{k-1})$.
- The party commits block $B_{k''}$ and all its ancestors.

(4) **Prepare.** Upon receiving $2t+1$ votes for block B_k, the leader forms certificate $C_v(B_k)$ and broadcasts a request $\langle \text{prepare}, C_v(B_k)\rangle_{L_v}$ to all parties. ▷ Executed by the leader

(5) **Vote2.** Upon receiving $\langle \text{prepare}, C_v(B_k)\rangle_{L_v}$, a party updates their lock to B_k and the locked certificate to $C_v(B_k)$. It sends $\langle \text{vote2}, C_v(B_k), v\rangle$ to L_{v+1}. ▷ Executed by all parties

Figure 1: View protocol for parties in view v.

(1) **Set timers.** Upon entering view v, where $(v \bmod t+1) = 0$, a party sets a view-k timer to expire at predetermined slots $k\tau$, for $k = 1 \ldots (t+1)$. It then proceeds to View Protocol for view v.

(2) **Timer expiration.** Upon timer expiration, the party stops processing messages and voting for view v.
- If $(v \bmod t+1) \neq 0$ (a non "epoch-view"), it executes View Protocol.
- **Epoch synchronization.** Else, if $v \bmod (t+1) = 0$, it performs an "epoch-view" procedure for view v:
  - send a timeout message $\langle \text{wish}, v+1\rangle$ to the $t+1$ view leaders in the epoch
  - any one of the $t+1$ leaders that collects $2t+1$ $\langle \text{wish}, v+1\rangle$ messages forming a TC_{v+1}, or obtains TC_{v+1}, broadcasts the TC to all parties.

(3) **Advance.** At any time, a party enters view v' where $v' > v$
- upon receiving a $TC_{v'}$ from any of the $t+1$ leaders for view v', where $v' \bmod (t+1) = 0$. In this case, a party also relays the TC to $t+1$ leaders in the epoch of v'.
- upon receiving $C_{v'-1}(C_{v'-1}(B_{k'}))$

Figure 2: Pacemaker protocol.

correct parties to respond, and then proceed to propose. The leader does not need to convince parties about the safety of its proposal by sending all status certificates it has received – this is because it provides enough time for all parties to report their highest lock; it will attach the highest certificate among all honest parties.

We emphasize that since Case 2 view-synchronization is invoked after a $\Omega(\Delta)$ delay, (asymptotically) there is no loss in latency.

In summary, the HotStuff-2's view-change mechanism is a simple addition over a vanilla two-phase HotStuff protocol, which incurs an extra 2Δ delay on top of $\tau = O(\Delta)$ under pessimistic conditions. In fact, most of the protocol pseudo-code below constitutes a vanilla two-phase HotStuff.

4.1 Performance Measures

We analyze the performance measures after GST.

Communication complexity. Observe that, when rotating leaders round-robin, in the worst case, we would have an honest leader within $t+1$ views. In each view, all messages are either leader-to-all or all-to-leader. Moreover, since we use threshold signatures, each message consists of $O(1)$ words. Thus, the total communication within a view is $O(n)$ words, resulting in total communication of $O(n^2)$ words. Moreover, for $t+1$ views, the communication complexity of the RareSync/Lewis-Pye pacemaker protocol is $O(n^2)$.

In the optimistic case, when the leader is honest, a block is committed within a view, incurring only $O(n)$ words of communication.

Worst-case latency. Again, in the worst-case, we would reach a view with an honest leader after $t + 1$ views. Moreover, each view incurs $O(\Delta)$ time. Thus, the worst-case commit latency is $O(n\Delta)$ time.

Optimistic responsiveness with a sequence of honest leaders. After GST, when we have a sequence of honest leaders, observe that all protocol steps (except the first honest leader's proposal in the sequence) are a result of obtaining a quorum of messages from the previous step, making the protocol optimistically responsive.

4.2 Security Proof

We introduce the notion of *direct* and *indirect* commits to aid our proof. We say a block B_k is committed in view v *directly* if it receives $C_v(C_v(B_k))$. We say a block is committed *indirectly* if this condition does not apply to a block in the view but to one of its successors.

Lemma 4.1. *If a party directly commits block B_k in view v, then a certified block that ranks no lower than $C_v(B_k)$ must equal or extend B_k.*

Proof. Suppose we consider a block $B'_{k'}$ that is certified in view v' to produce certificate $C_{v'}(B'_{k'})$. $C_{v'}(B'_{k'})$ has a rank higher than $C_v(B_k)$ if $v' > v$. We will use induction on view v'.

For the base case, we have $v = v'$. Thus, block $B'_{k'}$ is certified in view v. However, this requires votes from $2t + 1$ parties which is not possible due to a quorum intersection argument in view v.

For the inductive step, since $C_v(C_v(B_k))$ exists, it must be the case that a set S of $\geq 2t + 1$ parties have access to $C_v(B_k)$ and are locked on B_k or a block that extends B_k at the end of view v. By the inductive hypothesis, any certified block that ranks equally or higher from view v up to v' either equals or extends B_k. Thus, at the end of view v', the parties in set S are still locked on B_k or a block that extends B_k. Consider a proposal in view $v' + 1$. If the leader makes a proposal $B'_{k'}$ containing a certificate that does not extend B_k, by the inductive hypothesis, this certificate must be ranked lower than $C_v(B_k)$. Consequently, no honest party in set S will vote for it, and thus a certificate $C_{v'+1}(B'_{k'})$ cannot be formed. Thus, if $B'_{k'}$ was committed and $C_{v'+1}(C_{v'+1}(B'_{k'}))$ is formed it must be the case that $B'_{k'}$ extends B_k. □

Theorem 4.2 (Safety). *Two parties commit the same block B_l for each height l.*

Proof. Suppose for contradiction, two distinct blocks B_l and B'_l are committed at height l. Supposed B_l is committed as a result of B_k being directly committed in view v and B'_l is committed as a result of $B'_{k'}$ being directly committed in view v'. Thus, B_k is or extends B_l and B'_k is or extends $B'_{l'}$. WLOG, suppose $v \leq v'$. If $v = v'$, suppose, $k \leq k'$. By Lemma 4.1, $C_{v'}(B'_{k'})$ must equal or extend B_k. Thus, $B_l = B'_l$. □

Theorem 4.3 (Liveness). *Assuming all honest parties are in view v for time $> 6\Delta$ and the latest honest party enters view v within Δ time after the leader L_v, GST occurred more than Δ time before the earliest honest party entered view v, and leader L_v is honest, honest parties in the protocol would commit a block in view v.*

Proof. Let us consider the time period after GST has already occurred. Observe that the pacemaker protocol [9, 18] ensures that parties enter a view within Δ time of the view's leader; thus, if we set τ appropriately, the honest parties would overlap for $> 6\Delta$ time in view v. Consider an honest leader L_v's state at the start of its view. It could have entered the view using one of the two methods:

(1) **Using a TC.** In this case, the leader waits for 2Δ time and then proceeds to propose. Suppose the leader enters view v at time t. Observe that, all honest parties would enter view v no later than time $t + \Delta$. Hence, at time $t + \Delta$, they would send the highest certified block known to them to the leader L_v.
Since L_v waits for 2Δ time, it would receive locked blocks with certificates from all honest parties. Thus, the leader has access to the highest certified block across all honest parties. Hence, the leader's proposal would extend this highest certified block. Consequently, all honest parties can vote on this block.
(2) **Using $C_{v-1}(C_{v-1}(B_{k-1}))$.** In this case, the leader sends a proposal $\langle \text{propose}, B_k, v, C_{v-1}(B_{k-1}), C_{v-1}(C_{v-1}(B_{k-1}))\rangle_{L_v}$. On receiving $C_{v-1}(C_{v-1}(B_{k-1}))$, an honest party would immediately advance to view v and proceed to the vote step. Moreover, a proposal in view v cannot include a lock ranked higher than a view $v - 1$ certificate. Thus, all honest parties, on receiving a proposal containing this lock, would vote for the block.

In both cases, when the leader is honest, subsequent steps such as prepare and vote2 would proceed as expected since the message would contain a certificate from view v. In all, obtaining the highest certificate takes up to 2Δ time, proposal and vote takes up to 2Δ time, and sending prepare and vote2 consumes up to 2Δ time. Thus, the leader drives a commit within 6Δ time. □

REFERENCES

[1] Ittai Abraham, Guy Gueta, and Dahlia Malkhi. Hot-stuff the linear, optimal-resilience, one-message BFT devil. *CoRR*, abs/1803.05069, 2018.
[2] Mark Abspoel, Thomas Attema, and Matthieu Rambaud. Malicious security comes for free in consensus with leaders. *Cryptology ePrint Archive*, 2020.
[3] Marcos Kawazoe Aguilera and Sam Toueg. A simple bivalency proof that t-resilient consensus requires t+ 1 rounds. *Information Processing Letters*, 71(3-4):155–158, 1999.
[4] Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, University of Guelph, 2016.
[5] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.
[6] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in Constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.
[7] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.
[8] Jing Chen and Silvio Micali. Algorand, 2016.
[9] Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, and Manuel José Ribeiro Vidigueira. Byzantine consensus is θ (nˆ 2): The dolev-reischuk bound is tight even in partial synchrony! Technical report, Dagstuhl Publishing, 2022.
[10] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *Journal of the ACM (JACM)*, 32(1):191–204, 1985.
[11] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.
[12] Eli Gafni. Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing (PODC '98)*, 1998.
[13] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus

with asynchronous fallback. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers*, pages 296–315. Springer, 2022.

[14] Neil Giridharan, Heidi Howard, Ittai Abraham, Natacha Crooks, and Alin Tomescu. No-commit proofs: Defeating livelock in bft. *Cryptology ePrint Archive*, 2021.

[15] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: a scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 568–580. IEEE, 2019.

[16] Mohammad M Jalalzai, Jianyu Niu, Chen Feng, and Fangyu Gai. Fast-hotstuff: A fast and resilient hotstuff protocol. *arXiv preprint arXiv:2010.11454*, 2020.

[17] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 45–58, 2007.

[18] Andrew Lewis-Pye. Quadratic worst-case message complexity for state machine replication in the partial synchrony model. *CoRR*, abs/2201.01107, 2022.

[19] J-P Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.

[20] Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine View Synchronization. *Cryptoeconomic Systems*, 1(2), oct 22 2021. https://cryptoeconomicsystems.pubpub.org/pub/naor-cogsworth-synchronization.

[21] Espresso Systems. *https://hackmd.io/@EspressoSystems/EspressoSequencer*, 2022.

[22] The Diem Team. *https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf*, 2021.

[23] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.