Sync HotStuff: Simple and Practical Synchronous State Machine Replication

Ittai Abraham[1], Dahlia Malkhi[2]*, Kartik Nayak[3]*, Ling Ren[4]*and Maofan Yin[5]*

[1]VMware Research iabraham@vmware.com
[2]Calibra dahliamalkhi@gmail.com
[3]Duke University kartik@cs.duke.edu
[4]University of Illinois at Urbana-Champaign renling@illinois.edu
[5]Cornell University tedyin@cs.cornell.edu

***Abstract*—Synchronous solutions for Byzantine Fault Tolerance (BFT) can tolerate up to minority faults. In this work, we present Sync HotStuff, a surprisingly simple and intuitive synchronous BFT solution that achieves consensus with a latency of 2Δ in the steady state (where Δ is a synchronous message delay upper bound). In addition, Sync HotStuff ensures safety in a weaker synchronous model in which the synchrony assumption does not have to hold for all replicas all the time. Moreover, Sync HotStuff has optimistic responsiveness, i.e., it advances at network speed when less than one-quarter of the replicas are not responding. Borrowing from practical partially synchronous BFT solutions, Sync HotStuff has a two-phase leader-based structure, and has been fully prototyped under the standard synchrony assumption. When tolerating a single fault, Sync HotStuff achieves a throughput of over 280 Kops/sec under typical network performance, which is comparable to the best known partially synchronous solution.**

I. Introduction

Byzantine Fault Tolerance (BFT) protocols relying on a synchrony assumption have the advantage of tolerating up to one-half Byzantine faults [1], while asynchronous or partially synchronous protocols tolerate only one-third [2]. On the flip side, synchronous protocols are often considered impractical for three main reasons. First, existing synchronous protocols require a large number of rounds. Second, most synchronous protocols require lock-step execution (i.e., replicas must start and end each round at the same time), making them hard to implement and further exacerbating the latency problem. Third, an adversary may attack the network to violate the synchrony assumption, causing the protocol to be unsafe.

In this work, we introduce Sync HotStuff, a synchronous BFT state machine replication (SMR) protocol that addresses the above concerns with a surprisingly simple and intuitive solution (see Figure 1). In Sync HotStuff, in the standard synchrony model, a leader broadcasts a proposal; the replicas echo it; and each replica can commit after waiting for the maximum round-trip delay unless it hears by that time an equivocating proposal signed by the leader. (If the leader does not propose, replicas time out and perform a leader change; details on this step are given in the body of the paper.)

Simple yet powerful, Sync HotStuff achieves the following desirable properties. First, as in most synchronous solutions, Sync HotStuff tolerates up to one-half Byzantine replicas. Second, inspired by Hanke et al. [3], Sync HotStuff does not require lock-step execution in the steady state. Third, with minor modifications, Sync HotStuff can handle a weaker and more realistic synchrony model suggested by Chan et al. [4]. Finally, Sync HotStuff is prototyped and shown to offer practical performance. It achieves a throughput comparable to partially synchronous protocols and the commit latency is roughly a single maximum round-trip delay. Given the above properties, we believe Sync HotStuff can be the protocol of choice for single-datacenter replicated services as well as consortium blockchain applications.

We proceed to elaborate on the key techniques and key results of Sync HotStuff, which removes performance barriers on synchronous BFT under weaker assumptions.

Near-optimal latency. The first key contribution is the aforementioned extremely simple steady state protocol (Figure 1). We observe that waiting for a single maximum round-trip delay suffices for replicas to commit. Furthermore, our protocol does not have to be executed in a lock-step fashion, despite relying on synchrony. In other words, other than the concurrent waiting step, replicas move to the next step upon receiving enough messages of the previous step, without waiting for the conservative synchrony delay bound. This gives a latency of $2\Delta + O(\delta)$ in steady state where Δ denotes the known bound assumed on maximal network transmission delay and δ denotes the actual network delay, which can be much smaller than Δ.

Assuming $\delta \ll \Delta$, the above latency is within a factor of two of the optimal latency that can be obtained by synchronous protocols: we give a minor adaptation to the proof of Dwork et al. [2] to show that a Δ latency is necessary for any protocol tolerating more than one-third Byzantine faults. The Δ latency lower bound should not be surprising because a protocol that commits faster than Δ, in a way, does not take advantage of synchrony and is thus subject to the one-third partial synchrony barrier. In fact, we conjecture a stronger latency lower bound of 2Δ. Our intuition is that replicas can be out-of-sync by Δ, so one Δ is needed for lagging replicas to catch up and another Δ is needed for messages to be delivered (the current lower bound proof only captures the latter Δ).

*Part of the work was done while the authors were at VMware Research.

DOI 10.1109/SP40000.2020.00044

Moreover, though $O(\delta)$ latency is impossible to *guarantee* under more than one-third faults, it can be achieved optimistically. The Thunderella protocol [5] achieves $O(\delta)$ latency when the leader is honest and more than three-quarter of the replicas are responding. We show that our protocol can be adapted to incorporate this idea.

Practical throughput. The key technique to improve throughput is to move the *synchronous waiting steps* off the critical path. The only step in steady state that requires waiting for a conservative $O(\Delta)$ time is to check for a leader equivocation before committing and it is made concurrent to main logic. Thus, replicas start working on the next block without waiting for the previous blocks to be committed. (The non-blocking commit also reduces latency, since a block can now be proposed before the previous block is committed.) The other synchronous waiting steps are in the view-change protocol, which occurs infrequently. Therefore, in the steady state, replicas are always sending protocol messages and utilizing the entire network capacity, thus behaving exactly like partially synchronous protocols. Our experiments validate that Sync HotStuff achieves throughput comparable to partially synchronous protocols. In fact, since a synchronous solution tolerates more corruption (half vs. one-third), it requires fewer replicas to be deployed to tolerate a given number of faults. In our experiments, we observe that in some cases, Sync HotStuff can even slightly outperform partially synchronous solutions in throughput.

Safety despite some sluggish honest replicas. Synchronous protocols proven secure under the standard synchrony assumption fail to provide safety if a single message between honest replicas is delayed. Recently, Guo et al. [6] proposed a "weak synchrony" model that allows the message delay bound Δ, at any point in time, to be violated for a set of honest replicas. We call these replicas *sluggish*. We call the remaining honest replicas *prompt* and messages of *prompt* replicas can reach each other within Δ time. To reflect reality and be more conservative, the model allows sluggish replicas to be arbitrarily *mobile*, i.e., an adversary decides which replicas are sluggish at any time. Messages sent by or sent to a sluggish replica may take an arbitrarily long time *until* the replica is prompt again. Since "weak synchrony" has been used in the literature to describe other models (e.g., [7], [8], [9]), we will refer to this model as the *mobile sluggish* model in this paper. We call the synchrony model without mobile sluggish faults *standard synchrony*.

With standard synchrony, if a replica sends a message to another replica, it is guaranteed to arrive within Δ time. Our protocol and proofs crucially use this fact to achieve safety. With a mobile sluggish fault model, on the other hand, the delivery is not guaranteed if the sender or the receiver is sluggish. In that sense, the guarantee for a single replica sending or receiving a message is similar to that of partially synchronous network model. The central observation enabling us to tackle mobile sluggish faults is the following: assuming a minority of the replicas are sluggish or Byzantine at any point in time, if a replica receives a message from a majority of replicas, at least one of the senders must be prompt and honest. We use this observation atop Sync-HotStuff-under-standard-synchrony to obtain a protocol in the mobile sluggish model. The resulting protocol ensure safety as long as the number of sluggish plus Byzantine faults combined is less than one-half; in other words, at any time, a majority of replicas must be honest and prompt, which has been shown to be a necessary condition [6] in the mobile sluggish model.

Organization. In the remainder of this section, we define state machine replication. In Section II, we describe Sync HotStuff in the standard synchrony model without sluggish faults. Section III augments this protocol to tolerate sluggish faults. Section IV adds an optimistically responsive mode to Sync HotStuff with sluggish faults. Section V presents the results based on our implementation and evaluation. Section VI compares with closely related works.

A. Definitions and Model

State Machine Replication (SMR). A state machine replication protocol is used for building a fault-tolerant service to process client requests. The service consists of n replicas, up to f of which may be faulty. The service commits client requests into a linearizable log and produces a consistent view of the log akin to a single non-faulty server. More formally, a state machine replication service provides the following two guarantees:

> **(safety)** non-faulty replicas do not commit different values at the same log position,
> **(liveness)** each client request is eventually committed by all non-faulty replicas.

We assume that the network consists of pairwise, authenticated communication channels between replicas. We assume digital signatures and a public-key infrastructure (PKI), and use $\langle x \rangle_p$ to denote a message x signed by replica p. (It is sufficient to sign the hash digest of a message for efficiency.) Wherever it is clear from context, we omit the subscript p. We also assume that there is no drift between the clocks used by the replicas, i.e., the clocks run at the same rate. Our protocol is secure under a sluggish mobile adversary. However, for ease of exposition we first explain the protocol in the standard synchrony model. We describe these models in the respective sections.

II. Sync HotStuff under Standard Synchrony

We first present Sync HotStuff in the standard synchrony model (without mobile sluggish faults). Here, the synchrony assumption states that a message sent at time t by any replica arrives at another replica by time $t + \Delta$ where Δ is a known maximum network delay. We use δ to denote the actual message delay in the network. We show a protocol that tolerates minority Byzantine replicas, i.e., $n = 2f + 1$. We reiterate that although the protocol assumes synchrony, replicas do not progress in lock-steps.

Sync HotStuff takes the Paxos/PBFT's approach of having a *stable leader* in a steady state. Each period of steady state is called a *view*, numbered by integers. The leader of view v can simply be replica $(v \mod n)$, i.e., leaders are scheduled in a round-robin order. The leader is expected to keep making progress by committing client requests at increasing *heights*. If replicas detect Byzantine behavior by the leader or lack of progress in a view, they *blame* the leader and engage in a *view-change* protocol to replace the leader. Figures 1 and 2 describe the steady state and view-change protocols, respectively.

Blocks and block format. As commonly done in SMR, client requests are batched into *blocks*. The protocol forms a chain of blocks. We refer to a block's position in the chain as its *height*. A block B_k at height k has the following format

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

where b_k denotes a proposed value at height k and $H(B_{k-1})$ 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. 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. If a block B_k is an ancestor of another B_ℓ ($\ell \geq k$), we say B_ℓ *extends* B_k. Note that a block extends itself. We say two blocks B_ℓ and $B'_{\ell'}$ *equivocate* each other if they do not extend one another.

Certified and locked blocks. A key ingredient of BFT solutions is a *quorum certificate*, a set of signed votes on a block from a quorum of replicas in the same view. In Sync HotStuff, a quorum consists of $f + 1$ replicas (out of $2f + 1$). If a block B_k has a quorum certificate from view v, we say it is a *certified block* in view v, and write it as $\mathcal{C}_v(B_k)$.

Certified blocks are ranked first by views and then by heights, i.e., (i) blocks certified in a higher view has higher rank, and (ii) for blocks certified in the same view, a higher height gives a higher rank. During the protocol execution, each replica keeps track of all certified blocks and keeps updating the highest certified block to its knowledge.

In Sync HotStuff, replicas will *lock on* certified blocks at the beginning of each view. Looking ahead, the notion of locked blocks will be used to guard the safety of a commit.

Block chaining. Blocks across heights are chained by hashes (cf. block format) and certificates (cf. Figure 1). This idea originated from the Bitcoin white paper [10] and it was incorporated into BFT by Casper [11] and HotStuff [12]. It greatly simplifies BFT protocols since now the voting step on a block also serves as a voting step for all its ancestor blocks that have not been committed. Hence, crucially, committing a block commits all its ancestors.

A. Steady State Protocol

The steady state protocol runs in iterations. We explain each step in a iteration.

Propose. The leader L proposes a block $B_k = (b_k, H(B_{k-1}))$ by broadcasting $\langle \mathsf{propose}, B_k, v, \mathcal{C}_v(B_{k-1}) \rangle_L$. The proposal contains a view-v certificate for its predecessor block B_{k-1}. The first view-v certificate will be obtained in the view-change protocol in Section II-B. If any replica receives propose (or new-view , cf. Section II-B) messages containing equivocating blocks, we say the replica *detects leader equivocation*.

Vote. Each replica r, upon receiving the above proposal for B_k, broadcasts a vote $\langle \mathsf{vote}, B_k, v \rangle_r$ for B_k (Step 2), if it has not observed leader equivocation in the view. Note that r may first hear a proposal from a non-leader replica r' because r' forwards the proposal when voting. The voting step can thus be considered a *re-proposal* step.

Commit. Once replica r votes for a proposal for B_k, it starts a timer denoted $\mathsf{commit\text{-}timer}_{v,k}$ (Step 3). B_k is committed if r is still in view-v after 2Δ time (i.e., if r does not detect leader equivocation or a view change within 2Δ time). We note again that blocks across heights form a chain, and committing a block commits all its ancestors.

Note that the commit timers do not affect the critical path of progress. A replica votes and starts timers for subsequent heights without waiting for the previous height to be committed. In fact, a replica can potentially have many previous heights whose commit timers are still running.

Why does the 2Δ time ensure safety? Consider an honest replica r that votes for a block B_k at time t, does not observe leader equivocation or a view change, and hence commits B_k at time $t + 2\Delta$. We provide some intuition why this commit is safe by showing that B_k will be the only certified block at height k in the view. For that, we need to show that (i) B_k will be certified, and (ii) no equivocating block can be certified.

Replica r votes at time t. Its vote with the forwarded proposal reaches all honest replicas before $t + \Delta$. After that, honest replicas would not vote for an equivocating block. Thus, if any honest replica votes for an equivocating block, it must do so before $t + \Delta$; but then r would have detected leader equivocation before $t + 2\Delta$. This contradicts with the fact that r commits B_k. Hence, (ii) holds. The above also means all honest replicas will vote for B_k by time $t + \Delta$, and a certificate for B_k will form at all honest replicas before $t+2\Delta$. Hence, (i) holds. Note that (i) holds even if the leader did not propose B_k to all replicas.

Note that a commit by some honest replica at height k does not imply a commit by all honest replicas at that height. This is because a Byzantine leader can send equivocating proposals to a subset of honest replicas before their commit timers expires, causing them to not commit. Thus, to complete the safety proof, we also need to show that no honest replica will vote for equivocating blocks in subsequent views. This will be ensured by the view-change protocol in the next subsection.

B. View-change Protocol

The view-change protocol maintains safety across views and ensures liveness. The leader can prevent progress through two mechanisms – stalling and equivocating. The two conditions

Let v be the current view number and replica L be the leader of the current view. While in view v, a replica r runs the following protocol in the steady state.

1) **Propose.** If replica r is the leader L, upon receiving $\mathcal{C}_v(B_{k-1})$, broadcast $\langle \mathsf{propose}, B_k, v, \mathcal{C}_v(B_{k-1})\rangle_L$ where B_k extends B_{k-1}.
2) **Vote.** Upon receiving a proposal $\langle \mathsf{propose}, B_k, v, \mathcal{C}_v(B_{k-1})\rangle_L$ (not necessarily from L) where B_k extends B_{k-1}, if no leader equivocation is detected, forward the proposal to all other replicas, broadcast a vote in the form of $\langle \mathsf{vote}, B_k, v\rangle_r$, set $\mathsf{commit\text{-}timer}_{v,k}$ to 2Δ and start counting down.
3) **(Non-blocking) Commit.** When $\mathsf{commit\text{-}timer}_{v,k}$ reaches 0, commit B_k and all its ancestors.

Fig. 1: The steady state protocol under standard synchrony.

Let L and L' be the leaders of views v and $v+1$, respectively. Each replica r runs the following steps.

i **Blame and quit view.** If fewer than p proposals trigger r's votes in $(2p+4)\Delta$ time in view v, broadcast $\langle \mathsf{blame}, v\rangle_r$. Upon gathering $f+1$ $\langle \mathsf{blame}, v\rangle$ messages, broadcast them, and quit view v. If leader equivocation is detected, broadcast the two equivocating messages signed by L, and quit view.

ii **Status.** Wait for Δ time. Pick a highest certified block $\mathcal{C}_{v'}(B_{k'})$, lock on $\mathcal{C}_{v'}(B_{k'})$, send $\mathcal{C}_{v'}(B_{k'})$ to the new leader L', and enter view $v+1$.

iii **New-view.** The new leader L' waits for 2Δ time after entering view $v+1$ and broadcasts $\langle \mathsf{new\text{-}view}, v+1, \mathcal{C}_{v'}(B_{k'})\rangle_{L'}$ where $\mathcal{C}_{v'}(B_{k'})$ is a highest certified block known to L'.

iv **First vote.** Upon receiving $\langle \mathsf{new\text{-}view}, v+1, \mathcal{C}_{v'}(B_{k'})\rangle_{L'}$, if $\mathcal{C}_{v'}(B_{k'})$ has a rank equal to or higher than r's locked block, forward $\langle \mathsf{new\text{-}view}, v+1, \mathcal{C}_{v'}(B_{k'})\rangle_{L'}$ to all other replicas and broadcast $\langle \mathsf{vote}, B_{k'}, v+1\rangle_r$.

Fig. 2: The view-change protocol under standard synchrony.

of quitting a view, based on no progress and equivocation, defend against these two attacks, respectively (Step i). A leader is expected to propose a block every 2Δ time: one Δ for its proposal to reach other replicas and one Δ for other replicas' votes to arrive. This forces a Byzantine leader to propose a block every 2Δ time to avoid being overthrown. If a leader equivocates, i.e., sends propose or new-view messages that contain equivocating blocks, the equivocating messages serve as a proof of Byzantine behavior and, once received, make an honest replica quit the view. Note that equivocating messages may be received directly from L or forwarded by other replicas.

Once a replica quits view v, it stops voting in that view and aborts all commit-timers of that view. The replica then waits for Δ time, picks a highest certified block, locks on it, reports the lock to L', and enters the new view (Step ii).

The Δ wait before locking ensures that every honest replica learns all blocks committed by all honest replicas in previous views before sending its lock status to the new leader. This maintains the safety of all committed blocks by all honest replicas up until this view (formalized in Lemma 1).

Once in the new view, the new leader L' waits for 2Δ time to collect locked blocks from all honest replicas and then broadcasts a new-view message containing a highest certified block known to it (Step iii). When a replica receives a new-view message, if the certified block it contains ranks greater than or equal to its own locked block, it forwards the new-view message and broadcasts a vote for the block (Step iv).

C. Safety and Liveness

We say a block B_k is committed *directly* in view v if it is committed as a result of its own $\mathsf{commit\text{-}timer}_{v,k}$ expiring. We say a block B_k is committed *indirectly* if it is a result of directly committing a block B_ℓ $(\ell > k)$ that extends B_k.

Lemma 1. *If an honest replica directly commits B_ℓ in view v, then (i) no equivocating block is certified in view v, and (ii) every honest replica locks on a certified block that ranks equal to or higher than $\mathcal{C}_v(B_\ell)$ before entering view $v+1$.*

Proof. Suppose an honest replica r directly commits B_ℓ in view v at time t. Then, at time $t-2\Delta$, replica r votes for and forwards a proposal for B_ℓ. All honest replicas receive the proposal for B_ℓ by time $t-\Delta$.

For part (i), observe that after time $t-\Delta$, no honest replica will vote for an equivocating block in the same view. If any other honest replica had voted for an equivocating block $B'_{\ell'}$ before $t-\Delta$, replica r would have received the equivocating propose or new-view message for $B'_{\ell'}$ before time t, which contradicts the hypothesis of r committing B_ℓ directly in view v at time t. Therefore, an equivocating block will not get any honest vote in view v and will not be certified in view v.

For part (ii), let us understand the situation of honest replicas at time $t-\Delta$. First, no honest replica has quit view v by time $t-\Delta$ because otherwise r would have quit view v by time t and would not have committed B_ℓ in view v at time t. Second, since r entered view v before time $t-2\Delta$, every

honest replica has entered view v before time $t - \Delta$. Thus, every honest replica will vote for B_ℓ by time $t - \Delta$ and all honest replicas receive $\mathcal{C}_v(B_\ell)$ by time t. Due to the Δ wait before entering the next view, every honest replica enters view $v + 1$ after time t. Hence, before entering view $v + 1$, they will lock on $\mathcal{C}_v(B_\ell)$ or higher. □

Lemma 2 (Unique Extensibility)**.** *If an honest replica directly commits B_ℓ in view v, then any certified block that ranks equal to or higher than $\mathcal{C}_v(B_\ell)$ must extend B_ℓ.*

Proof. Let S be the set of certified blocks that rank equal to or higher than $\mathcal{C}_v(B_\ell)$ but do not extend B_ℓ. Suppose for contradiction that $S \neq \emptyset$. Let $\mathcal{C}_{v^*}(B_{k^*})$ be a lowest ranked block in S. It is easy to show that $v^* > v$; otherwise, we have $v^* = v$, $k^* \geq \ell$, and it contradicts Lemma 1(i). Also note that as B_{k^*} does not extend B_ℓ, its predecessor B_{k^*-1} does not extend B_ℓ, either.

For $\mathcal{C}_{v^*}(B_{k^*})$ to exist, some honest node must vote for B_{k^*} in view v^* upon receiving $\langle \mathsf{new\text{-}view}, v^*, \mathcal{C}_{v'}(B_{k^*})\rangle$ where $v' < v^*$ or $\langle \mathsf{propose}, B_{k^*}, v^*, \mathcal{C}_{v^*}(B_{k^*-1})\rangle$. If it is the former, $\mathcal{C}_{v'}(B_{k^*})$ must rank higher than or equal to $\mathcal{C}_v(B_\ell)$, because due to Lemma 1(ii) and the fact that a replica never decreases the rank of its lock, every honest replica locks on $\mathcal{C}_v(B_\ell)$ or higher at the beginning of view v^*. However, this contradicts the definition of $\mathcal{C}_{v^*}(B_{k^*})$, because $\mathcal{C}_{v'}(B_{k^*})$ now belongs to S and ranks lower than $\mathcal{C}_{v^*}(B_{k^*})$. A similar contradiction exists for the latter case: $\mathcal{C}_v(B_{k^*-1})$ belongs to S and ranks lower than $\mathcal{C}_{v^*}(B_{k^*})$. □

Theorem 3 (Safety)**.** *No two honest replicas commit different blocks at the same height.*

Proof. Suppose for contradiction that two distinct blocks B_k and B'_k are committed at height k. Suppose B_k is committed as a result of B_ℓ being directly committed in view v and B'_k is committed as a result of $B'_{\ell'}$ being directly committed in view v'. This implies B_ℓ extends B_k and $B'_{\ell'}$ extends B'_k. Without loss of generality, assume $v \leq v'$; if $v = v'$, further assume $\ell \leq \ell'$. By Lemma 2, $B'_{\ell'}$ extends B_ℓ. Thus, $B'_k = B_k$. □

Lemma 4 (Liveness)**.** *All honest replicas keep committing new blocks.*

Proof. Due to the blame condition, a Byzantine leader needs to propose at least p proposals that trigger honest replica votes within $(2p+4)\Delta$ time to avoid a view-change (for all $p > 0$). All honest replicas will soon commit these proposed blocks unless a view-change occurs. Due to the round-robin leader order, eventually there will be an honest leader.

Next, we will show that once the leader is honest, a view-change will not occur and all honest replicas keep committing new blocks. The leader may enter the view Δ later than others and need to wait for 2Δ time before proposing. Another Δ is needed for others to receive the proposal. The 2Δ wait ensures an honest leader receives locked blocks of all honest replicas up until the beginning of that view. Hence, the block it proposes will extend the locked blocks of all honest replicas and receive votes from all honest replicas. After that, the honest leader is able to propose a block every 2Δ time: one Δ for its proposed block to reach all honest replicas and another Δ for all honest replicas' votes to arrive. Thus, an honest leader is able to make every other honest replica vote for p proposals in $(2p + 4)\Delta$ time. In addition, an honest leader does not equivocate. So no honest replica will blame the honest leader and all honest replicas keep committing new blocks. □

D. Efficiency Analysis

Throughput. In steady state (Figure 1), the key step that uses the synchrony bound Δ is the commit step. But as we have mentioned, the commit step is not on the critical path (non-blocking). Thus, the choice of Δ, no matter how conservative, does not affect the protocol's throughput in steady state. Thus, Sync HotStuff should have similar throughput as partially synchronous protocols. Our experiments in Section V confirm this.

Latency. From an honest leader's perspective, each block incurs a latency of $2\Delta+\delta$ after being proposed. (Step 1 proceeds at the actual network delay δ.) But for SMR, it is more customary to calculate latency from a client's perspective, that is, the time difference between when a client sends a request and when it receives a response. If a client's request arrives between two leader proposals, it incurs an additional delay before being getting proposed. From a client's perspective, it takes δ to send its request to the replicas; on average, this request will arrive half way between two leader proposals, which are 2δ time apart, so another δ delay on average; it takes an additional δ time for replicas to reply to the client. So the average client latency of Sync HotStuff is $2\Delta + 4\delta$. Our experiments in Section V confirm this.

For comparison, the best prior synchronous protocol in terms of latency is Hanke et al. [3]. Its average latency is $8\Delta + 9\delta$ from a leader's perspective [13], and $9\Delta + 11.5\delta$ from a client's perspective, following a similar analysis.

E. Bound on Responsiveness

Sync HotStuff commits with a 2Δ latency in the steady state when $f < n/2$. For completeness, in this section, we show a lower bound on the latency when $f > n/3$. The lower bound and the proof closely follow Dwork et al. [2]. For clarity, we present the bound in the Byzantine broadcast formulation. Recall that in Byzantine broadcast, a designated *sender* tries to broadcast a value to n parties. A solution needs to satisfy three requirements:

> **(termination)** all honest parties eventually commit,
> **(agreement)** all honest parties commit on the same value, and
> **(validity)** if the sender is honest, then all honest parties commit on the value it broadcasts.

Theorem 5. *There exists no Byzantine broadcast protocol that simultaneously satisfy the following:*

- *termination, agreement and validity as defined above;*

- *tolerates $f \geq n/3$ Byzantine faults;*
- *terminates in less than Δ time if the designated sender is honest.*

Proof. Suppose such a protocol exists. Divide parties into three groups P, Q and R, each of size at most f. Consider the following three scenarios. In Scenario A, parties in Q are Byzantine and remain silent and an honest designated sender sends 0. In this scenario, parties in P and R commit 0 in less than Δ time. In Scenario B, parties in P are Byzantine and remain silent and an honest designated sender sends 1. In this scenario, parties in Q and R commit 1 in less than Δ time. In Scenario C, the designated sender is Byzantine and sends 0 to P and 1 to Q; parties in R are Byzantine and they behave like in Scenario A to P, and behave like in scenario B to Q. Messages from P take Δ to reach Q and messages from Q take Δ to reach P. Before time Δ, P receive no messages from Q and Q receive no messages from P, and thus the scenario is indistinguishable from Scenario A to P and indistinguishable from Scenario B to Q. Thus, P commits 0 and Q commits 1 in less than Δ time, violating agreement. □

III. Sync HotStuff with Mobile Sluggish Faults

The standard synchrony model used in the previous section requires that every message sent by an honest replica arrives at every other honest replica within Δ time. In practice, such an assumption may not hold all the time due to potential unforeseen aberrations in the network at either the sender or the receiver, causing some messages to be delayed. Under such aberrations, a protocol proved secure under the standard synchrony assumption may lose safety. For our protocol specifically, if a replica that voted for an equivocating block runs into a network glitch, then another honest replica may not receive it in time and may incorrectly commit another block. A potential way to "fix" this is to account for the sender (or receiver) of the delayed message as Byzantine and thus tolerate fewer actual Byzantine faults. Unfortunately, over the course of a long execution, every replica is bound to observe such an aberration and this "fix" will result in a dishonest majority of replicas, thus breaking safety eventually.

A. The Mobile Sluggish Model

Chan et al. [4] consider a weaker model that allows some replicas to be *sluggish*, i.e., allows delays for messages sent/received by sluggish replicas in the network. On the other hand, messages sent by *prompt* replicas will respect the synchrony bound. More specifically, if a replica r_1 is prompt at time t_1, then any message sent by r_1 at time $\leq t_1$ will arrive at a replica r_2 prompt at time t_2 if $t_2 \geq t_1 + \Delta$. Moreover, the set of sluggish replicas can arbitrarily change at every instant of time. Hence, we call this model the mobile sluggish fault model. We denote the number of sluggish replicas by d, the number of Byzantine replicas by b and the total number of faults by f. Thus, $f = d + b$.

We note that the mobile sluggish model expects that a message sent by a sluggish replica would respect the synchrony bound as soon as it becomes prompt. In practice, this model captures temporary loss in network connectivity causing message delays. The replica can resend messages or download buffered messages as soon as network connectivity is restored. However, it is not a good model for capturing a replica going offline for a while since this model would require the replica to either buffer a huge amount of messages to be resent or to resend each message many times, both of which are impractical.

Guo et al. [6] show that no protocol can tolerate a total number of faults (sluggish plus Byzantine replicas) greater than $n/2$. The intuition is that a majority set consisting of Byzantine and sluggish replicas might reach a commit decision without interacting with the rest of the world and might cause conflicting commits. Again, we assume $n = 2f + 1$. Thus, we assume $f + 1$ replicas are honest and prompt at any time. Moreover, our protocol provides liveness when $f + 1$ honest replicas are prompt for a "sufficiently long" period of time, i.e., there can be sluggish replicas but they are not mobile. The duration is directly related to the time required to commit a block in the protocol.

B. Protocol

In the synchronous protocol described in Section II, the 2Δ period after a vote ensures two things: (i) every honest replica receives $\mathcal{C}_v(B_k)$ within before entering the next view, and (ii) no honest replica votes for an equivocating block.

If this replica can be sluggish, unfortunately, neither of the above arguments holds. Other replicas may not receive the proposal it forwards and hence certificates may not form; and even if the leader equivocates, this sluggish replica may not know about it in time.

The following modifications are used to ensure the above two properties in the presence of mobile sluggish faults. To ensure (i), we require a replica to start its timer after receiving $\mathcal{C}_v(B_k)$ (contained in the next proposal) from $f + 1$ replicas. One of those replicas must be honest and prompt when the timer was started, so all prompt replicas receive $\mathcal{C}_v(B_k)$ in time. For (ii), we require a replica to commit only after hearing from $f + 1$ replicas that no equivocation happened in a 2Δ period. This is safe because an equivocation could not have missed all of them. We call the former step a *pre-commit* and the latter step a commit.

Interestingly, despite a weaker model than standard synchrony, based on the intuition presented above, the protocol is only marginally different from the one in Section II. For clarity we present the entire steady state protocol and gray out the repetition in Figure 3. The view-change protocol need not change.

C. Safety and Liveness

The proof in the mobile sluggish model has the same structure as the standard synchrony model. Lemma 6 below is almost identical to Lemma 1 except that the claim is made for $f + 1$ honest replicas instead of all honest replicas, which is as expected because the remaining d honest replicas can be sluggish. We can prove unique extensibility and safety

Let v be the current view number and replica L be the leader of the current view. While in view v, a replica r runs the following protocol in the steady state.

1) **Propose.** If replica r is the leader L, upon receiving $\mathcal{C}_v(B_{k-1})$, broadcast $\langle \mathsf{propose}, B_k, v, \mathcal{C}_v(B_{k-1})\rangle_L$ where B_k extends B_{k-1}.
2) **Vote.** Upon receiving a proposal $\langle \mathsf{propose}, B_k, v, \mathcal{C}_v(B_{k-1})\rangle_L$ (not necessarily from L) where B_k extends B_{k-1}, if no leader equivocation is detected, forward the proposal to all other replicas, broadcast a vote in the form of $\langle \mathsf{vote}, B_k, v\rangle_r$.
3) **(Non-blocking) Pre-commit.** On receiving $\langle \mathsf{propose}, B_{k+1}, v, \mathcal{C}_v(B_k)\rangle_L$ from $f+1$ replicas, set $\mathsf{pre\text{-}commit\text{-}timer}_{v,k}$ to 2Δ and start counting down. When $\mathsf{pre\text{-}commit\text{-}timer}_{v,k}$ reaches 0, pre-commit B_k and broadcast $\langle \mathsf{commit}, B_k, v\rangle_r$.
4) **(Non-blocking) Commit.** On receiving $\langle \mathsf{commit}, B_k, v\rangle$ from $f+1$ replicas, commit B_k and all its ancestors.

Fig. 3: The steady state protocol with mobile sluggish faults.

identically as Lemma 2 and Theorem 3 by invoking Lemma 6 in place of Lemma 1.

As before, we say a block B_k is committed *directly* if it is committed due to $f+1$ pre-commits. We say a block B_k is committed *indirectly* if it is a result of directly committing a block extending B_k.

Lemma 6 (Lemma 1 extended to mobile sluggish)**.** *If an honest replica directly commits B_ℓ in view v, then (i) no equivocating block is certified in view v, and (ii) $f+1$ honest replicas lock on a certified block that ranks equal to or higher than $\mathcal{C}_v(B_\ell)$ before entering view $v+1$.*

Proof. If an honest replica directly commits B_ℓ in view v, then $d+1$ honest replicas pre-commit B_ℓ in view v. Denote the set of these $d+1$ replicas by R. Let the earliest pre-commit among R be performed by replica r_1 at time t. At time $t-2\Delta$, replica r_1 must have received view-v proposals for $B_{\ell+1}$ from $f+1$ distinct replicas. One of those replicas, say replica r_2, is honest and prompt at time $t-2\Delta$. Denote the set of honest and prompt replicas at time $t-\Delta$ by R'. Every replica R' receives $\langle \mathsf{propose}, B_{\ell+1}, v, \mathcal{C}_v(B_\ell)\rangle_L$ from r_2 by time $t-\Delta$. We will prove that the set R' is the required set that does not vote for equivocating blocks in view v and locks on a certified block that ranks equal to or higher than $\mathcal{C}_v(B_l)$ before entering view $v+1$.

For part (i), observe that after time $t-\Delta$, replicas in R', having seen a proposal for $B_{\ell+1}$, will not vote for a block that equivocates B_ℓ. If any replica in R' had voted for an equivocating block before $t-\Delta$, its broadcast of the equivocating propose or new-view message will reach all honest replicas that are prompt at time t by time t. At least one replica in R would be prompt at time t. This replica would have detected leader equivocation by time t and would not have pre-committed B_ℓ, a contradiction. Therefore, an equivocating block will not get any vote from R' in view v and will not be certified in view v.

For part (ii), observe that no replica in R' has quit view by time $t-\Delta$, because otherwise some replica in R would not have pre-committed following the same argument as above. Therefore, every replica in R' locks on $\mathcal{C}_v(B_\ell)$ or higher before quitting view v, which is before entering view $v+1$. □

Remark. Note that the proof of the above lemma shows that all replicas in R' receive $\mathcal{C}_v(B_k)$ before quitting view v. Thus, the Δ wait during view-change is not needed for the protocol with sluggish faults.

Liveness. In the mobile sluggish model, liveness is guaranteed only during periods in which $f+1$ honest replicas including the leader stay prompt. In that case, the same arguments in Lemma 4 hold. We do not repeat the proof.

D. Efficiency

In the mobile sluggish model, each pre-commit timer starts 3δ later. The commit messages add another round of communication and δ latency. So the total latency becomes $2\Delta + 4\delta + 4\delta = 2\Delta + 8\delta$ from the client's perspective.

IV. Sync HotStuff with Optimistic Responsiveness

In this section, we incorporate the Thunderella [5] optimistic responsive mode into Sync HotStuff. In Section II, a certificate/quorum required only $f+1$ votes. Thus, a vote from a single honest replica *can* result in a certificate if all f Byzantine replicas vote on the same block. Therefore, before committing a block, a replica needs to wait long enough to hear all honest replicas' votes and make sure none of them voted for an equivocating block. The commit latency thus inherently depends on the maximum network delay Δ. In contrast, partially synchronous protocols rule out the existence of a conflicting certificate with larger quorums. For instance, PBFT requires $> 2n/3$ votes (in two phases) and tolerates $f < n/3$ Byzantine replicas. A simple quorum intersection argument shows that two equivocating blocks cannot both receive $> 2n/3$ votes. Thus, partially synchronous protocols commit as soon as these quorums of votes are obtained, so the latency does not depend on Δ.

Pass and Shi [5] use the term *responsive* to capture the above latency distinction. A protocol is said to be *responsive* if the latency only depends on the actual network δ but not the maximum network delay Δ. A protocol is said to be *optimistically responsive* if it achieves responsiveness when some additional constraints are met.

Since Sync HotStuff aims to tolerate up to minority corruption, similar to Thunderella [5], to achieve responsiveness the

Let v be the current view number and replica L be the leader of the current view. While in view v, a replica r runs the following protocol in the steady state. All certificates created in view v require $> 3n/4$ votes.

1) **Propose.** If replica r is the leader L, upon receiving $\mathcal{C}_v(B_{k-1})$, broadcast $\langle \mathsf{propose}, B_k, v, \mathcal{C}_v(B_{k-1})\rangle_L$ where B_k extends B_{k-1}.
2) **Vote.** Upon receiving $\langle \mathsf{propose}, B_k, v, \mathcal{C}_v(B_{k-1})\rangle_L$ (not necessarily from L) where B_k extends B_{k-1}, if no leader equivocation is detected, forward the proposal to all other replicas, and broadcast a vote in the form of $\langle \mathsf{vote}, B_k, v\rangle_r$.
3) **Pre-commit.** On receiving $\langle \mathsf{propose}, B_{k+1}, v, \mathcal{C}_v(B_k)\rangle_L$ from $f+1$ replicas, pre-commit B_k and broadcast $\langle \mathsf{commit}, B_k, v\rangle_r$ right away.
4) **Commit.** On receiving $\langle \mathsf{commit}, B_k, v\rangle$ from $f+1$ replicas, commit B_k and all its ancestors.

Fig. 4: The steady state protocol in a responsive view.

Let v be the current view and L be the current leader. Let L_2 be the next leader and v_2 be the synchronous view of L_2. Note that $v_2 = v+2$ if v is a synchronous view, and $v_2 = v+1$ is v is a responsiveness view.

i **Blame.** If fewer than p proposals trigger r's votes in $(2p+4)\Delta$ time in view v, broadcast $\langle \mathsf{blame}, v_2 - 1\rangle_r$. Upon gathering $f+1$ $\langle \mathsf{blame}, v_2-1\rangle$ messages, broadcast them along with $\langle \mathsf{blame2}, v_2-1\rangle_r$, and quit view v. If leader equivocation is detected, broadcast $\langle \mathsf{blame2}, v_2-1\rangle_r$ and the two equivocating messages, and quit view v.

ii **Status.** Upon gathering $f+1$ $\langle \mathsf{blame2}, v_2-1\rangle$ messages, wait for 2Δ time. Pick a highest certified block $\mathcal{C}_{v'}(B_{k'})$, lock on $\mathcal{C}_{v'}(B_{k'})$, send $\mathcal{C}_{v'}(B_{k'})$ to the new leader L_2, and enter view v_2.

iii **New-view.** The new leader L_2 waits for 2Δ time after entering view v_2 and broadcasts $\langle \mathsf{new\text{-}view}, v_2, \mathcal{C}_{v'}(B_{k'})\rangle_{L_2}$ where $\mathcal{C}_{v'}(B_{k'})$ is a highest certified block known to L_2.

iv **First vote.** Upon receiving $\langle \mathsf{new\text{-}view}, v_2, \mathcal{C}_{v'}(B_{k'})\rangle_{L_2}$, if $\mathcal{C}_{v'}(B_{k'})$ has a rank equal to or higher than r's locked block, forward $\langle \mathsf{new\text{-}view}, v_2, \mathcal{C}_{v'}(B_{k'})\rangle_{L_2}$ to all other replicas, broadcast $\langle \mathsf{vote}, B_{k'}, v_2\rangle_r$.

Fig. 5: The view-change protocol to support responsive reviews.

quorum size should be $> 3n/4$. This will give Sync HotStuff *a responsive mode* when messages from $> 3n/4$ replicas reach within time δ. This happens if the *actual* number of faults is less than $n/4$. Put differently, if more than $n/4$ (but fewer than $n/2$) replicas are faulty, they can prevent responsiveness but they cannot cause a safety violation. In that case, we fall back to the *synchronous mode* in Section III.

Protocol. The new protocol will give two views to each leader. Each odd view is a synchronous view and runs the protocol in Figure 3. Each even view is a responsive view and runs the protocol in Figure 4. As mentioned, a certificate from a responsive view requires a quorum of $> 3n/4$. The only other difference from Figure 3 is that a replica pre-commits B_{k-1} immediately on receiving $f+1$ proposals for B_k, rather than waiting for a 2Δ period.

To switch from a synchronous view to a responsive view, the leader simply broadcasts a signed switch message. Upon receiving the switch message, a replica forwards it and enters the responsive view. The leader sends a new-view message to obtain the first certificate in the view as before.

Whenever a view fails due to an equivocating leader or lack of progress, replicas engage in a view-change protocol to move to the next leader's synchronous view. Guarding the safety of responsive commits across view changes requires some care as we explain below. In the protocol without responsiveness, a commit implied that a certificate was broadcast by some prompt replica 2Δ time earlier. This ensured that a certificate was obtained at a majority of honest replicas before they quit the view. Since there is no 2Δ time in the responsive mode, the above does not hold. The solution is to insert a 2Δ wait between quitting the old view and entering the new view (Step ii in Figure 5). The delay is introduced at a replica *after learning that a majority of replicas have quit the view*, giving sufficient time for the certificates to be received at a majority of prompt replicas before they enter the next view. These changes will be utilized in the proof of Lemma 7.

A. Safety of Responsive Views

Lemma 7 (Lemma 6 extended to responsiveness views)**.** *If an honest replica directly commits B_ℓ in view v, then (i) no equivocating block is certified in view v, and (ii) $f+1$ honest replicas lock on a certified block that ranks equal to or higher than $\mathcal{C}_v(B_\ell)$ before entering view $v+1$.*

Proof. If view v is a synchronous view, then the proof of Lemma 6 still applies. It remains to prove the lemma for responsive views.

Part (i) directly follows from quorum intersection. Since certificates in a responsive view requires $> 3n/4$ votes, for there to be equivocating certified blocks, at least $3n/4+3n/4-n = n/2 > f$ replicas need to vote for equivocating blocks, which cannot happen.

Part (ii) mostly follows from the proof of Lemma 6. Suppose an honest replica directly commits B_ℓ in view v. Then a set R of $d+1$ honest replicas have pre-committed B_ℓ. Let the earliest pre-commit among them be performed at time t. This implies that more than $f+1$ replicas broadcast a proposal containing $\mathcal{C}_v(B_\ell)$ before time t. At least one of them is honest and prompt at time t. Denote the set of honest and prompt replicas at time $t+\Delta$ by R'. R' receives $\mathcal{C}_v(B_\ell)$ by time $t+\Delta$. We will now prove that the set R' satisfies part (ii). It remains to show that no replica in R' has entered view $v_2 = v+1$. Suppose this is not true, i.e., one of the replicas in R', say replica r' has entered view v_2 before time $t+\Delta$. In that case, due to the 2Δ wait during view-change, r' has received $f+1$ blame2 messages before time $t-\Delta$. Thus, $f+1$ replicas have sent blame2 and quit view v before time $t-\Delta$. At least one of them is honest and prompt at time $t-\Delta$. At least one replica in the pre-committing set R would be prompt at time t and would have received this blame2 message by time t. It would have prevented the pre-commit of B_ℓ at that replica, a contradiction. □

The rest of the safety proof remains unchanged.

V. Evaluation

In this section, we first evaluate the throughput and latency of Sync HotStuff under different parameters and conditions (batch size, payload, and client command load). We then evaluate the impact of Δ on throughput and latency and show it is insignificant as expected. Lastly, we compare Sync HotStuff with HotStuff [12] and Dfinity [3].

A. Implementation Details and Methodology

We implement the Sync HotStuff protocol under the standard synchrony model. Our implementation is an adaptation of the open-source implementation of HotStuff [12]. We modify the HotStuff code to primarily replace the core protocol logic while reusing some of its utility modules, such as its event queue and network library.

In our implementation, each block contains a batch of commands sent by clients. A command consists of a unique identifier and an associated payload. We refer to the maximum number of commands in a block as the batch size. A conceptual representation of a block is shown in Figure 6.

All throughput and latency results were measured from clients which are separate processes running on machines different from those for replicas. Each client generates a number of outstanding commands and broadcasts them to every replica. Replicas only use the unique identifier (e.g. hash) of a command to represent it in proposals and votes. To execute the commands for the replicated state machine, a replica either has the command content received from the client's initial broadcast, or fetches it from the leader if the client crashes before it finishes the broadcast. We use four machines, each running four client processes, to inject commands into the system. Each client process can maintain a configurable number of outstanding commands at any time.

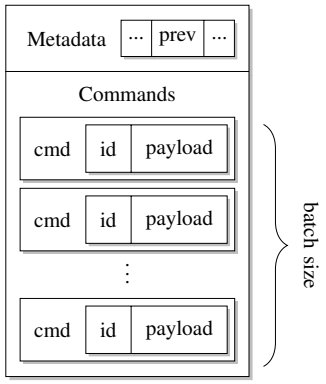


Fig. 6: Structure of a block in the implementation.

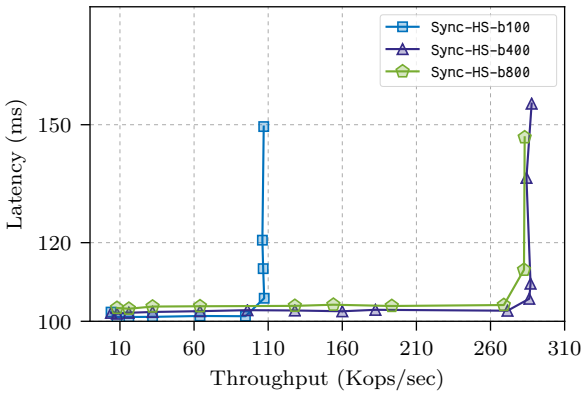
We ensure that the performance of replicas will not be limited by lack of client commands.

Experimental setup. All our experiments were conducted over Amazon EC2 where each replica was executed on a `c5.4xlarge` instance. Each instance had 16 vCPUs supported by Intel Xeon Platinum 8000 processors. All cores sustained a Turbo CPU clock speed up to 3.4GHz. The maximum TCP bandwidth measured by `iperf` is around 4.9 Gbps, i.e., 0.6 Gigabytes per second. We did not throttle the bandwidth in any run. The network latency between two machines is measured to be less than 1 ms. We used `secp256k1` for digital signatures in votes and a certificate consists of a compact array of `secp256k1` signatures.

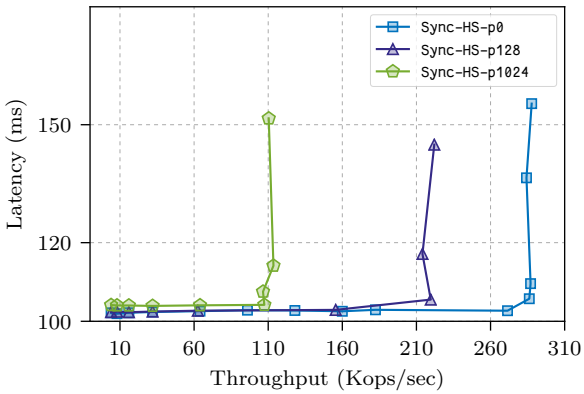
Baselines. We compare with two baselines: (i) HotStuff, a partially synchronous protocol, and (ii) Dfinity , a synchronous protocol. We use HotStuff as a baseline because Sync HotStuff shares the same code base as HotStuff enabling a fair comparison, and because HotStuff achieves comparable (or even better) performance to state-of-the-art partially synchronous BFT implementation [12]. We pick Dfinity as our other baseline because it is the state-of-the-art synchronous BFT protocol. We did not find an implementation of Dfinity's cousensus protocol in its Github repository, so we implemented our own version of Dfinity with our codebase (which should also help ensure a fair comparison). While implementing and evaluating Dfinity, we made several simplifications that are favorable to Dfinity. For instance, it was shown that a malicious leader can exploit a flaw in the original Dfinity design to force unbounded communication complexity [13]. We did not implement the suggested fix to this flaw (and of course we did not exploit this flaw). We assume all leaders in Dfinity are honest, which will improve Dfinity's theoretical latency from 9Δ to 7Δ. We simulate their Verifiable Random Functions (VRF), by essentially assuming VRF generation takes negligible time in Dfinity. Implementing these extra fixes and actual mechanisms will only further hurt Dfinity's performance.

B. Basic Performance

We first evaluate the basic performance of Sync HotStuff to tolerate $f = 1$ fault for a synchrony bound of $\Delta = 50$ ms.

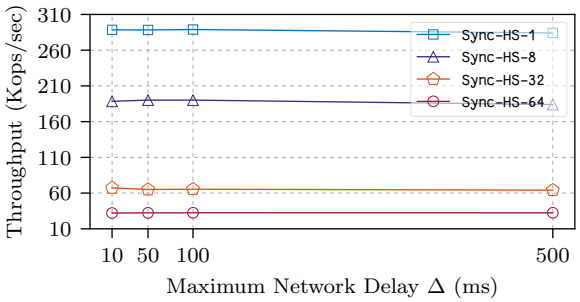


(a) Varying batch sizes.

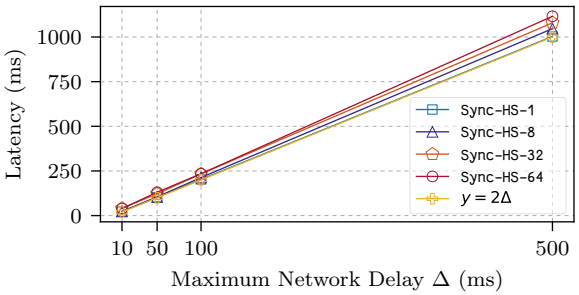


(b) Varying payload.

Fig. 7: Throughput vs. latency of Sync HotStuff at varying batch sizes and payloads at $\Delta = 50$ ms and $f = 1$.



(a) Δ vs. throughput.



(b) Δ vs. latency.

Fig. 8: Performance of Sync HotStuff at varying Δ and f at batch size = 400 and 0/0 payload.

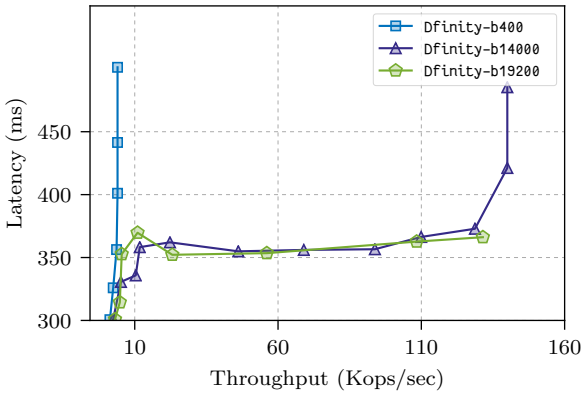
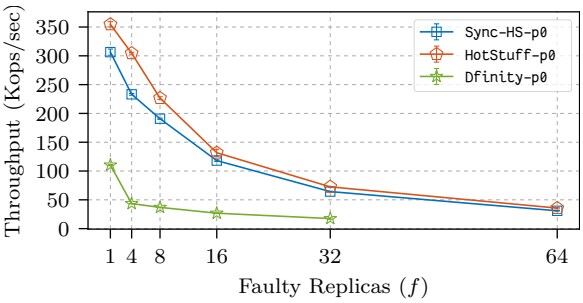


Fig. 9: Throughput vs. latency of Sync HotStuff at varying batch sizes at $\Delta = 50$ ms and $f = 1$ for Dfinity [13].

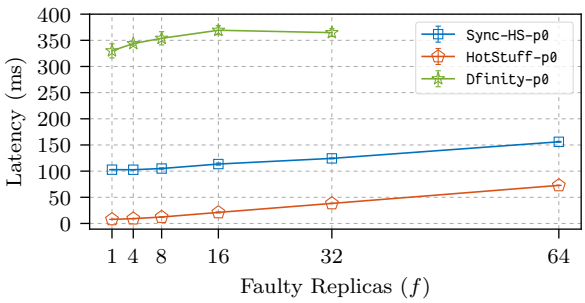
We measure the observed throughput (in number of commands committed/sec, or ops/sec) and end-to-end latency for clients (in ms). We conduct two experiments. The first one fixes the payload and varies batch size (Figure 7a) while the other fixes a batch size and varies the payload size (Figure 7b).

In Figure 7a, each command has a zero-byte payload to demonstrate the overhead induced solely by consensus and state machine replication. We consider three different batch sizes, 100, 400 and 800, represented by the three lines in the throughput-latency graph. In the graph, each point represents the measured throughput and latency for one run with a given "load" by the clients. More specifically, a client process maintains a fixed number of outstanding commands at any moment. When an outstanding command is committed, a new command is immediately issued to keep up with the specified number. We vary the size of the outstanding command pool to simulate different loads. The points at the lower left represent the state when the system is not saturated by client commands. As the load increases, the throughput initially increases without incurring a loss in latency. Finally, after the load saturates the bandwidth, the throughput remains unchanged (or slightly degrades) when clients inject more commands, while the latency goes up. The latency increases because the commands stay in the command pool longer before they are proposed in a block for consensus. For a batch size of 400, we observe that the throughput is saturated at around 280 Kops/sec. There is no further throughput gain when batch size increases from 400 to 800. So in all of our following experiments, we fix our batch size to 400.

We also test how payload size of a command affects performance. Figure 7b shows the performance with three client request/reply payload sizes (in bytes) 0/0, 128/128 and 1024/1024, denoted by "`p0`", "`p128`", and "`p1024`". In addition to the actual payload, each command also contains an 8 byte counter to differentiate the commands. For example,

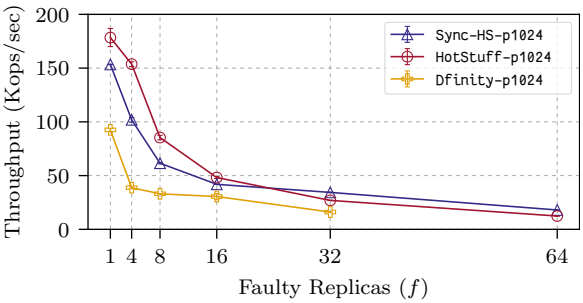


(a) f vs. throughput.

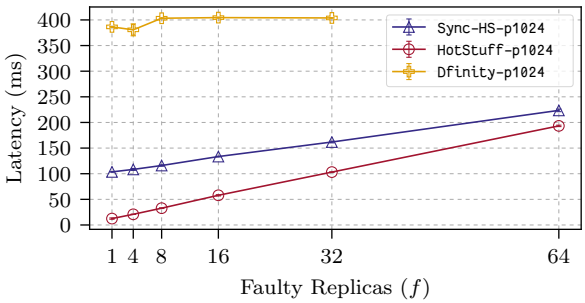


(b) f vs. latency.

Fig. 10: Performance as function of faults at Δ = 50 ms, optimal batch size, and 0/0 payload.



(a) f vs. throughput.



(b) f vs. latency.

Fig. 11: Performance as function of faults at Δ = 50 ms, optimal batch size, and 1024/1024 payload.

the actual command size for 0/0 is 8 bytes.

C. The Impact of Δ on Performance

In the steady state of Sync HotStuff, replicas advance to the next step as soon as previous messages arrive, without waiting for any conservative Δ bound. Thus, although each block still incurs 2Δ latency to be committed, the system is able to move on after a single round-trip time, process new blocks in pipeline, and saturate available network bandwidth. Figures 8a and 8b study the effect of varying Δ on throughput and latency. Each line represents a choice of f, denoted by "1", "8", "32", "64". As expected, we observe that the saturated throughput remains unaffected by different choices of Δ, whereas the latency deviates little from the theoretical 2Δ line. We do note that the latency remains unaffected *only* when the Δ bound is conservative, because that is when the time for certifying a block (the $O(\delta)$ terms in our theoretical analysis) is overshadowed by the 2Δ wait. When tolerating a larger number of faults or when deployed on slower network conditions (e.g., consortium blockchains), Δ should be set appropriately to ensure safety.

D. Scalability and Comparison with Prior Work

We perform an experiment to understand how Sync HotStuff scales as the number of replicas increases. We also compare this with HotStuff and Dfinity. In our baseline, clients issue zero-byte payload commands and saturate the system, without overloading the replicas. We then vary the choice of f. Each experiment is repeated five times with the same setup to average out fluctuations. A data point shows the average value, capped by error bars indicating the standard deviation. Since synchronous protocols tolerate one-half faults as against one-third in case of partial synchrony, for the same f, the actual number of replicas is $2f + 1$ for Sync HotStuff and Dfinity, whereas it is $3f + 1$ for HotStuff. We would also like to point out that such comparison is not entirely fair since HotStuff does not assume synchrony. Nevertheless, it is still helpful to understand the performance of Sync HotStuff by comparing it to a state-of-the-art partially synchronous protocol like HotStuff.

Comparison with HotStuff. Figures 10 and 11 show the throughput and latency for two different payload configurations, 0/0 and 1024/1024. We use a batch size of 400 for Sync HotStuff and HotStuff. Generally, the throughput of Sync HotStuff tends to be slightly worse than HotStuff. But at more faults, the throughput of Sync HotStuff gets closer to HotStuff and in the 1024/1024 case, eventually surpasses HotStuff. This is because in both cases the system is bottlenecked by a leader communicating with all other replicas and since Sync HotStuff requires fewer replicas to tolerate f faults, its performance scales better than HotStuff.

Comparison with Dfinity. For Dfinity, we first perform an experiment to determine good batch sizes that maximize its throughput. The results are shown in Figure 9. We observe that Dfinity requires a batch size of 14000 to reach its peak throughput of ∼130 Kops/sec. The reason why Dfinity requires a much larger batch size is because proposals are made much

less frequently at every 2Δ time. In contrast, in Sync HotStuff, a new proposal is made every $2\delta \ll 2\Delta$ time, i.e., as soon as the previous proposal has been "processed" (i.e., certified). This allows Sync HotStuff to fully utilize available network bandwidth with much smaller batch sizes.

Figures 10b and 11b show the latency for two different payload configurations, 0/0 and 1024/1024. As can be seen in the figures, the latency of Dfinity varies between 330ms and 400ms. This is much higher than that for Sync HotStuff and is consistent with the expected theoretical average latency as described in Section V-A. We also observe that at $f = 64$, the large batch size we choose for Dfinity violates our $\Delta = 50$ms synchrony bound, leading to safety violations. Hence, our evaluation does not include that data point.

VI. Related Work

Several decades of research on the Byzantine agreement problem [14] brought a myriad of solutions. Dolev and Strong gave a deterministic protocol for its related problem, Byzantine broadcast, with the tolerance of $f < n$ [15]. Their protocol achieves $f + 1$ round complexity and $O(n^2 f)$ communication complexity. The $f + 1$ round complexity matches the lower bound for deterministic protocols [16], [15]. To further improve round complexity, randomized protocols have been introduced [17], [18], [19], [20], [21], [22]. We review the most recent and closely related works below.

Some key design goals of Sync HotStuff are inspired by recent related works. In particular, elimination of lock-step synchrony is first explored by Hanke et al. [3] and the mobile sluggish model is introduced by Guo et al. [6]. Compared to these works, Sync HotStuff uses techniques that are significantly simpler and more efficient to achieve the same goals.

Dfinity. The Dfinity Consensus protocol described in [3] is a replication protocol in the synchrony model that tolerates $f < n/2$ Byzantine faults. It makes a key observation that a synchronous replication protocol can start processing the next client request without waiting for the previous one to commit. While a standard practice in partially synchronous protocols, this was not obvious for synchronous protocols.

However, it was later discovered [13] that the presentation in [3] allowed unbounded communication complexity due to an exploitable requirement that honest replicas vote for all leader proposals, including equivocating ones from the same leader. Another inefficient design in Hanke et al. is that each leader makes only one proposal before getting replaced by a new random leader. This hurts latency because (i) up to half of the proposal opportunities are wasted on Byzantine leaders, (ii) their leader election step is on the critical path and is blocking (in this sense, Hanke et al. did not fully remove lock-step execution.) This results in a large latency of $9\Delta + O(\delta)$.

In comparison, Sync HotStuff removes lock-step execution using a simple and natural protocol (replicas do not vote for equivocating proposals). Sync HotStuff embraces the stable leader approach, common in the partial synchrony SMR (e.g., PBFT [23], Paxos [24]), that uses a steady state leader to drive many decisions. These techniques allow Sync HotStuff to achieve $2\Delta + O(\delta)$ latency and quadratic communication complexity.

Guo et al. and PiLi. Guo et al. [6] introduced the mobile sluggish model (called weak synchronous model in that work). This model better reflects reality compared to the standard synchrony model and we adopt it. PiLi [4] presents a BFT SMR protocol in the mobile sluggish model. Its solution is theoretical and highly involved. It assumes lock-step execution in "epochs". Each epoch lasts for 5Δ time and the protocol commits five blocks after 13 consecutive epochs if certain conditions are met. Thus, PiLi requires a latency of at least 40Δ-65Δ (65Δ for the earliest and 40Δ for the latest of the five blocks). In contrast, we observe that simple techniques suffice to handle mobile sluggish faults at almost no extra overhead.

Thunderella. The notion of optimistic responsiveness (under one-quarter faults) was introduced in Thunderella [5]. Thunderella observes that it is safe to commit a decision in $O(\delta)$ time if $> 3n/4$ votes are received. In this paper, we adopt the key idea of Thunderella to achieve optimistic responsiveness. But we also make two changes. First, when more than $3n/4$ replicas are correct, Thunderella commits a decision after a single round of voting. However, the decision cannot be conveyed to external clients, hence it does not support SMR. Sync HotStuff uses two rounds to commit in the responsive mode, and hence provides safety for SMR. Second, Thunderella uses the synchronous mode to monitor the progress of the responsive mode and, if the responsive mode does not make progress quickly, falls back to the synchronous mode. The fallback mechanism is presented in a black-box fashion but it is unclear how it works in a non-Nakamoto-style protocol. In Sync HotStuff, we take the conventional approach of having replicas monitor the progress of the responsive reviews, and using the view-change protocol to perform the fallback.

XFT. A different type of protocol with optimistic responsiveness is XFT [25]. XFT guarantees responsiveness when a group of $f + 1$ honest replicas is determined. Thus, when the actual number of faults is t, it may take $\binom{n}{f+1}/\binom{n-t}{f+1}$ view-changes for an honest group of $f + 1$ to emerge; after that, the protocol is responsive. Such a solution is practical when t is a small constant but for $t = \Theta(n)$, it requires an exponential number of view-changes to find an honest group. In comparison, Sync HotStuff and Thunderella are responsive under $t < n/4$ faults after at most t view-changes.

VII. Conclusion

In this work, we introduce Sync HotStuff, a simple and practical synchronous BFT SMR protocol. Sync HotStuff does not require lock-step execution, tolerates mobile sluggish faults, and offers practical performance. As we mentioned, the mobile sluggish fault model captures short network glitches but is not ideal for replicas going offline for too long. It

remains interesting future work to come up with more realistic synchronous models as well as practical solutions in them.

Acknowledgment. We thank Atsuki Momose [26] for pointing out some mistakes in a previous version of this paper and suggesting a potential fix. We thank Zhuolun Xiang and Nibesh Shrestha for helpful feedback.

REFERENCES

[1] M. Fitzi, "Generalized communication and security models in byzantine agreement," Ph.D. dissertation, ETH Zurich, 2002.

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

[3] T. Hanke, M. Movahedi, and D. Williams, "Dfinity technology overview series, consensus system," *arXiv preprint arXiv:1805.04548*, 2018.

[4] T. H. Chan, R. Pass, and E. Shi, "Pili: An extremely simple synchronous blockchain," 2018.

[5] R. Pass and E. Shi, "Thunderella: Blockchains with optimistic instant confirmation," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 3–33.

[6] Y. Guo, R. Pass, and E. Shi, "Synchronous, with a chance of partition tolerance," 2019.

[7] R. Friedman and R. Van Renesse, "Strong and weak virtual synchrony in horus," in *Proceedings 15th Symposium on Reliable Distributed Systems*. IEEE, 1996, pp. 140–149.

[8] M. Biely, P. Robinson, and U. Schmid, "Weak synchrony models and failure detectors for message passing (k-) set agreement," in *International Conference On Principles Of Distributed Systems*. Springer, 2009, pp. 285–299.

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

[10] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[11] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *arXiv preprint arXiv:1710.09437*, 2017.

[12] M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham, "HotStuff: BFT consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019*, 2019, pp. 347–356.

[13] I. Abraham, D. Malkhi, K. Nayak, and L. Ren, "Dfinity consensus, explored," Cryptology ePrint Archive, Report 2018/1153, 2018.

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

[15] D. Dolev and H. R. Strong, "Authenticated algorithms for byzantine agreement," *SIAM Journal on Computing*, vol. 12, no. 4, pp. 656–666, 1983.

[16] M. J. Fischer and N. A. Lynch, "A lower bound for the time to assure interactive consistency," *Information processing letters*, vol. 14, no. 4, pp. 183–186, 1982.

[17] M. Ben-Or, "Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols," in *Proceedings of the second annual ACM symposium on Principles of distributed computing*. ACM, 1983, pp. 27–30.

[18] M. O. Rabin, "Randomized Byzantine generals," in *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*. IEEE, 1983, pp. 403–409.

[19] P. Feldman and S. Micali, "An optimal probabilistic protocol for synchronous byzantine agreement," *SIAM Journal on Computing*, vol. 26, no. 4, pp. 873–933, 1997.

[20] M. Fitzi and J. A. Garay, "Efficient player-optimal protocols for strong and differential consensus," in *Proceedings of the twenty-second annual symposium on Principles of distributed computing*. ACM, 2003, pp. 211–220.

[21] J. Katz and C.-Y. Koo, "On expected constant-round protocols for byzantine agreement," *Journal of Computer and System Sciences*, vol. 75, no. 2, pp. 91–112, 2009.

[22] I. Abraham, S. Devadas, D. Dolev, K. Nayak, and L. Ren, "Synchronous byzantine agreement with expected $O(1)$ rounds, expected $O(n^2)$ communication, and optimal resilience," in *Financial Cryptography and Data Security (FC)*, 2019.

[23] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[24] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[25] S. Liu, C. Cachin, V. Quéma, and M. Vukolic, "XFT: practical fault tolerance beyond crashes," in *12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2016, pp. 485–500.

[26] A. Momose and J. P. Cruz, "Force-locking attack on sync hotstuff," Cryptology ePrint Archive, Report 2019/1484, 2019.