Arena: Multi-leader Synchronous Byzantine Fault Tolerance

Hao Lu
Zhejiang University
luhao@zju.edu.cn

Jian Liu✉
Zhejiang University
jian.liu@zju.edu.cn

Kui Ren
Zhejiang University
kuiren@zju.edu.cn

***Abstract*—Byzantine fault-tolerant state machine replication (BFT-SMR) replicates a state machine across a set of replicas, and processes requests as a single machine even in the presence of Byzantine faults. Recently, synchronous BFT-SMRs have received tremendous attention due to their simple design and high fault-tolerance threshold.**

In this paper, we propose Arena, the first *multi-leader* synchronous BFT-SMR. Thanks to the synchrony assumption, Arena gains the performance benefit from multi-leader with a much simpler design (compared to other partially synchronous multi-leader designs). Furthermore, it is more robust: "no progress" of a leader will not trigger a view-change. Our experimental results show that Arena achieves a peak throughput of up to 7.7× higher than the state-of-the-art.

1. Introduction

Byzantine fault-tolerant state machine replication (BFT-SMR) replicates a state machine across a set of replicas, and ensures that the service is both safe and live even in the presence of Byzantine faults. BFT-SMRs based on a synchrony assumption have the advantage of tolerating up to one-half Byzantine faults [1], but they are typically considered to be slow as their commit latency depends on the pessimistic bound of the network delay Δ. In comparison, asynchronous or partially synchronous BFT-SMRs are *responsive*, i.e., their commit latency only depends on the actual network delay δ. Nevertheless, synchronous BFT-SMRs have still received tremendous attention due to their simple and intuitive designs, and great breakthroughs have been made. For example, recent synchronous BFT-SMRs [2], [3], [4] allow the replicas to commit responsively (independent of Δ) when some optimistic conditions are met, a.k.a. *optimistic responsiveness*.

However, these synchronous BFT-SMRs rely on a single-leader to coordinate the protocol, which in turn enforces disproportionately high load on that leader. Furthermore, when the system is geo-replicated, clients might have to communicate with a remote server, which incurs additional overhead. In this paper, we aim to improve the performance of synchronous BFT-SMR by leveraging a *multi-leader* design, s.t., clients can choose which replica to send requests to and the loads are distributed evenly among all replicas.

A straightforward way to design a multi-leader BFT-SMR is to run multiple single-leader instances in parallel, each of which has a different leader. Most existing multi-leader designs [5], [6], [7], [8] follow this paradigm, but they are extremely complex due to their partial synchrony assumption. Furthermore, they require all replicas to vote for all replicas' proposals *separately*, which is a huge burden in both communication and computation (due to signature generation and verification). Interestingly, the synchrony assumption allows us to significantly simplify the multi-leader design. For example, we could have a replica vote for multiple proposals with a single vote, which is difficult to achieve in an asynchronous or partially synchronous network, because there is no guarantee when a replica can gather enough proposals.

Our contribution. In this paper, we propose Arena, which (to the best of our knowledge) is the *first* synchronous BFT-SMR. It has two prominent advantages over existing synchronous BFT-SMRs [2], [3], [4], [9], [10]:

- **Efficiency:** All replicas simultaneously propose requests so that the network bandwidth can be fully utilized. As a result, it achieves a much higher throughput.
- **Robust:** "No progress" of a leader will not trigger a view-change. In contrast, prior solutions need to run view-change when a leader either equivocates or stalls.

In more detail, we propose two versions of Arena. The first version simply has all replicas run in two Δ-epochs (in a steady-state): (1) gather proposals from other replicas in the first Δ-epoch, and (2) vote for the received proposals in the second Δ-epoch. Notice that if all non-crash replicas are honest and they start the protocol at the same time (i.e., synchronized starts), they will receive the same set of proposals and send the same vote. Once a replica receives different votes, it will initiate a view-change protocol, after which the *potentially* faulty replicas will be prohibited to make proposals. This aggressive view-change allows us to maintain safety for our extremely simple steady-state protocol. We further improve its performance by piggybacking its vote-epoch on the next propose-epoch. We name this version *pipelined* Arena.

The second version of Arena achieves optimistic responsiveness. In its fast path, a replica can vote directly upon

✉*Jian Liu is the corresponding author*

Notation	Description
R	Replica
n	number of replicas
f	number of faulty replicas
v	view number
b	a batch of proposed requests
B	a block of requests
$C(B)$	a commit certificate for B
$H()$	hash function
$Sig()$	signing function
σ	a signature
Σ	a set of signatures
Δ	pessimistic bound of the network delay
δ	actual network delay
L	leader for coordinating view-change

TABLE 1: Summary of frequent notations.

receiving all n proposals and commit directly upon receiving all n non-conflicting votes. That means it has a responsive commit latency of 2δ when no faults exists. In its slow path, a replica waits for 2Δ to gather proposals and waits for another Δ before sending its vote; it commits upon receiving non-conflicting votes from more than one-half replicas.

This is a common design for optimistic protocols. In scenarios like SMRs or consortium blockchains, no-fault is the commonest case. We name this version *optimistic* Arena. We remark that our optimistic Arena does not require an explicit switch between the two paths; instead, replicas exist in both paths simultaneously. Furthermore, it no longer needs to assume synchronized starts: it works as long as all honest replicas are initialized and start to propose within a time interval of Δ.

We evaluate Arena on a testbed consisting of 91 AWS VMs. Our experimental results show that Arena achieves a peak throughput of up to 819 708 TPS, 4.2× higher than Sync HotStuff [2], 3.6× higher than OptSync [4].

Organization. In the remainder of this paper, we first provide some essential definitions for synchronous BFT-SMRs in Section 2. In Section 3, we show the design overview of Arena. Then, we provided the details and proofs for both pipelined Arena and optimistic Arena in Section 4 and Section 5 respectively. Next, we extensively evaluate its performance and compare it to the state-of-the-art in Section 6. In the end, we survey related work in Section 7 and conclude the paper in Section 8. A summary of frequently used notations is listed in Table 1.

2. Model and Definitions

In this section, introduce some necessary concepts and definitions for this paper.

Byzantine fault-tolerant state machine replication (BFT-SMR). A BFT-SMR replicates a state machine across a set of n replicas Rs, and processes clients' requests as a single machine, even if f replicas behave arbitrarily (i.e., Byzantine faults). Typically, it runs as follows: (i) clients submit requests to replicas and wait for responses; (ii) replicas run a *consensus* protocol to agree on the order of request; (iii) replicas execute the operations in the requests following the agreed order; and (iv) replicas respond to the corresponding clients with the execution results. Clearly, the core component of a BFT-SMR is its consensus protocol, which ensures the following two properties:

- *Safety:* all non-faulty replicas execute the requests in the same order, a.k.a. *agreement*; each executed request was proposed by a client, a.k.a. *validity*.
- *Liveness:* a request proposed by a client will eventually be executed, a.k.a. *termination*.

Synchrony. We assume the communication between replicas is *synchronous*. Namely, messages between replicas may take at most Δ time to deliver; in other words, an adversary can delay a message for an arbitrary time upper bounded by Δ. However, the actual message delay is δ, which is much smaller than Δ. It is commonly known that under a synchrony assumption, a BFT-SMR of n replicas can tolerate $f < n/2$ Byzantine faults [1].

View-based execution. Most BFT-SMRs progress through a series of views that are sequentially numbered. Within each view, a.k.a. a *steady state*, a designated leader is expected to propose values and make progress by committing client requests at increasing heights. If replicas detect equivocation or lack of progress in the current view, they collectively perform a *view-change* to replace the faulty leader. It is worth mentioning that, in our proposed protocols, view-change only happens for equivocation, not for lack of progress of the protocol.

Blocks and block certificates. We have all n replicas propose requests in batches bs. The proposed requests are later combined into a block B_k pointing to its predecessor B_{k-1}, i.e., $B_k := (b_{k,1}||...||b_{k,n}, H(B_{k-1}))$, where $H()$ is a cryptographic hash function. Notice that a request will be removed from B_k if it conflicts with another request in front of it. We call a block's position in the chain as its *height*. We say B_k *extends* B_l $(k \geq l)$ if B_l is an ancestor of B_k; B_k and $B'_{k'}$ *equivocate* each other if they are not equal to and do not extend one another.

A *block certificate (a.k.a. commit certificate)* represents a set of signatures (or an aggregate signature) on a block by a quorum of replicas, denoted by $C_v^{\alpha}(B_k)$, where α is the quorum size and v is the view number. Whenever the quorum size is not important, we represent the certificate by $C_v(B_k)$ for simplicity. Replicas *lock on* certified blocks at the beginning of each view.

3. Design Overview

In this section, we show step-by-step how we design a multi-leader synchronous BFT-SMR.

3.1. **Pipelined** Arena

We start from an extremely simple design: we assume all honest replicas start the protocol at the same time i.e., synchronized starts (we will show how we get rid of this assumption later), and they run in *lock-steps*: start and end every Δ-epoch at the same time. The run the following two Δ-epochs:

- **Propose.** At the beginning of this epoch, each honest replica broadcasts a proposal, which includes a batch of requests together with a signature of this batch. The epoch is long enough (i.e., Δ) such that each replica is guaranteed to receive all proposals sent by honest replicas.
- **Vote.** At the beginning of this epoch, each honest replica combines the proposals (received during the propose-epoch) into a block and broadcasts a vote for this block. Again, at the end of this epoch, each replica is guaranteed to receive all votes sent by honest replicas.

A proposal is *valid* only if:

- it has been included in at least $(f+1)$ votes; **and**
- there is no equivocating proposal sent by the same replica.

Next, the requests inside the valid proposals are committed corresponding to the order of senders' IDs. In particular, a request will be neglected if it conflicts with another request in front of it. It is guaranteed that all honest replicas will commit the same set of requests in the same order.

Notice that if all non-crash replicas are honest (i.e., send the same proposal to all other replicas), they will receive the same proposals and send the same vote. On the other hand, a faulty replica can either (1) send different messages to different replicas, or (2) send the same message to a subset of replicas. Both of such misbehaviours can be caught in the end of the vote-epoch and trigger view-change. In case (1), these equivocating messages can serve as proof for Byzantine behaviors. Once received, an honest replica will trigger view-change. However, case (2) is tricky as we cannot determine who is cheating. For example, there are three replicas R_1, R_2, and R_3, which propose b_1, b_2 and b_3 respectively; if R_2 only votes for b_2 and b_3, we cannot determine whether this is because R_1 did not send b_1 to R_2, or because R_2 did not include b_1 in its vote. However, due to synchrony, if R_1 has broadcasted b_1, R_2 must have received it; therefore, either R_1 did not send the proposal to R_2 or R_2 did not include that proposal in its vote; they cannot both be honest. To this end, we take an aggressive approach by prohibiting both R_1 and R_2 to make proposals thereafter. We will prove the following Lemmas:

- All honest replicas ban the same replicas (cf. Lemma 6).
- Honest majority always holds among the replicas that can make proposals (cf. Lemma 7).

In the worst case, there will be one honest replica left and the protocol becomes a single-leader BFT-SMR. After a certain period, we could allow the prohibited replicas to make proposals again. We remark that this aggressive approach for banning replicas is necessary: if we do not ban any replica, the faulty replicas can make view-change keep happening, harming liveness; if we do not ban a pair of replicas, it is possible that there is a single faulty replica left in the end.

Recently, a pipelined-BFT paradigm was proposed to improve protocol performance: if a block requires two rounds of voting, we could piggyback the second round on the next block's voting. Although our protocol only requires one round of voting, we could still leverage it to propose new blocks. Namely, replicas can combine the vote-epoch of height-k with the propose-epoch of height-$(k+1)$. The communication pattern of this pipelined Arena is shown in Figure 1.

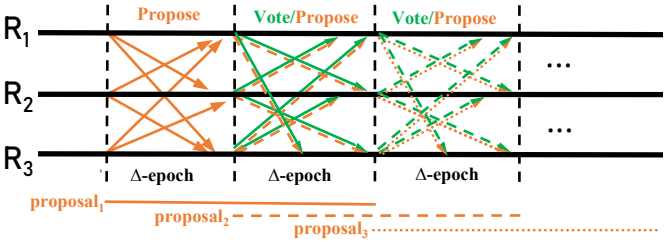


Figure 1: Communication patterns for pipelined Arena.

This multi-leader BFT-SMR has two prominent advantages over existing single-leader designs [2], [3], [4]:

1) Empirically, the block proposing phase is the most expensive phase of an SMR protocol. In a single-leader SMR, the throughput is limited by the outbound bandwidth of the proposing replica, resulting in wasted outbound bandwidth from other replicas. Conversely, multi-leader SMR allows for full utilization of all replicas' outbound bandwidth.
2) If honest replicas crashes or faulty replicas stalls, there are still at least $(f+1)$ honest replicas that will propose requests to keep the protocol running. In other words, view-change occurs only when there is a Byzantine leader. In contrast, in a single leader design, if the leader crashes or stalls, view-change needs to be triggered.

Next, we show how we further reduce its response latency.

3.2. **Optimistic** Arena

The protocol described in 3.1 has a commit latency that depends on the pessimistic network delay Δ. On the other hand, a protocol is *responsive* if its commit latency only depends on the actual network delay δ (rather than Δ). In this section, we seek to make Arena responsive when some optimistic conditions are met, i.e., *optimistic responsiveness* [2], [3], [4].

The key observation is that a replica can vote directly upon receiving $(2f+1)$ proposals (without having to wait until the end of the propose-epoch); similarly, it can commit directly upon receiving $(2f+1)$ non-conflicting votes

(without having to wait until the end of the vote-epoch). That means the responsive commit latency can be as small as 2δ.

However, after such responsive commits, replicas may end up making the next proposals at different time points, rendering our protocol incorrect. We tackle this issue in two directions: (1) limit such time discrepancies among honest replicas to Δ and (2) tolerate the Δ time discrepancy.

To limit the time discrepancies among honest replicas, we have each replica forward its $(2f+1)$ received votes immediately after a responsive commit. Suppose an honest replica responsively commits at t, then other honest replicas are able to commit before $t+\Delta$. Consequently, all replicas will start the next round within a time interval of Δ.

Nevertheless, due to this Δ time discrepancy, it can no longer guarantee that a replica will receive all proposals from honest replicas in the next Δ-epoch. Interestingly, we can resolve this problem by extending the duration of the propose-epoch from Δ to 2Δ. Specifically, each replica locally sets a propose-timer of 2Δ after sending a proposal; then, it is for sure that a replica will receive all proposals from honest replicas before propose-timer reaching 0, even though other honest replicas may send the proposals Δ later. A side benefit we get from this strategy is that we no longer need the assumption of synchronized starts; instead, we only need to assume that all honest replicas are initialized within a time interval of Δ.

Similar to the propose-epoch, we could extend the vote-epoch to 2Δ so that each replica can receive all votes from other honest replicas. However, this is an overkill and we would like to stick to the Δ-epoch for "vote". To achieve this, we borrow the idea from OptSync [4]: we have replicas wait for Δ before sending their votes. Specifically, in the slow path, each replica forwards its received proposals in an ack after the 2Δ propose-epoch, and then sets a vote-timer to Δ; a replica can vote only when its vote-timer reaching 0, thereafter it can commit the requests upon receiving $(f+1)$ non-conflicting votes. In the fast path, a replica can send the ack immediately after receiving $(2f+1)$ proposals and it can commit immediately after receiving $(2f+1)$ non-conflicting acks. In both paths, a replica needs to broadcast its commit certificate, i.e., $(2f+1)$ acks or $(f+1)$ votes, which can be combined with its next proposal. It is worth mentioning that the commit certificate can be an aggregate signature.

Figure 2 shows the communication patterns for both fast path and slow path. We remark that Arena replicas exist in both paths simultaneously without requiring an explicit switch. In contrast, the general strategy employed in the slow-path/fast-path paradigm [11] is to initially start in one of the two paths; when certain conditions are met, an explicit switch is performed to move to the other path. The explicit switch between the paths incurs a latency in such protocols.

4. Pipelined Arena in Detail

In this section, we provide a detailed description and correctness proof for pipelined Arena.

4.1. The protocol

We present the details of pipelined Arena in Figure 3.

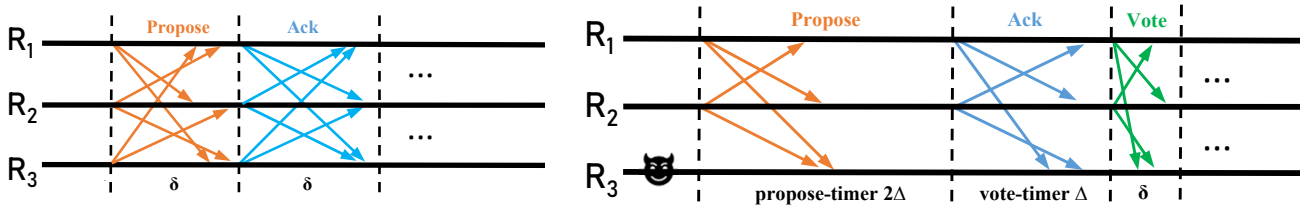
As we mentioned in Section 3.1, all replicas run in lock-steps: start and end every Δ-epoch at the same time. In the beginning of the kth epoch, each replica R_i proposes a batch of requests for height-k. To process requests in a pipelined manner, it also votes for height-$(k-1)$ and commits for height-$(k-2)$. The propose message is a batch of requests together with a signature on the batch: $\sigma_{k,i} := Sig(sk_i, H(b_{k,i})||v||k)$.

The vote message is a signature on the proposals to be voted: $\sigma'_{k-1,i} := Sig(sk_i, \sigma_{k-1,1}||...||\sigma_{k-1,n})$. It is important to include $\Sigma_{k-1,i} := \{\sigma_{k-1,1}, ..., \sigma_{k-1,n}\}$ in the vote message; otherwise, if inconsistent votes appear, an honest replica is not able to determine who is misbehaving. Specifically, if R_i receives two different votes from R_h and R_j, it can check $\Sigma_{k-1,h}$ and $\Sigma_{k-1,j}$ to see which σ_{k-1} leads to the inconsistency, and broadcasts the corresponding blame message.

To commit, R_i checks the $\geq (f+1)$ non-$\perp$ signatures in $\{\sigma'_{k-2,1}, ..., \sigma'_{k-2,n}\}$. If they are all consistent, R_i commits $B_{k-2} := (b_{k-2,1}||... ||b_{k-2,n}, H(B_{k-3}))$ and broadcasts a commit certificate $C_v^{f+1}(B_{k-2})$ that is computed as an aggregate signature of $\{\sigma'_{k-2,1}, ..., \sigma'_{k-2,n}\}$. Otherwise, R_i broadcasts a blame message signaling equivocation or missing proposals to trigger view-change. In fact, we could use a less aggressive approach for view-change here, i.e., they continue with the steady state as long as they receive $\geq (f+1)$ consistent votes, despite of other inconsistent votes.

After triggering the view-change, R_i waits for 2Δ before entering the next view, which is useful in proving Lemma 2. During this 2Δ, R_i keeps sending blames when receiving messages that can trigger "Blame and quit view". This is important to have all replicas receive the same set of blame messages so that they can determine the correctness of the new-view message.

For each view, we introduce a leader that is only for coordinating the view-change. Specifically, it is responsible for sending a new-view message that contains the latest commit certificate and a set of replicas to be banned to propose blocks. In particular, for "Quit view on detecting missing proposals", the sender R_j of the proposal and the sender R_h of the vote (that does not contain that proposal) will be banned. For example, R_i reports having received a proposal from R_j in its vote, while R_h's vote does not contain this proposal. As per the rule, R_j and R_h will be banned. Due to synchrony, if R_j has broadcasted a proposal, R_h must have received it. Therefore, either R_j did not send the proposal to R_h or R_h did not include that proposal in its vote. They cannot *both* be honest. If there are multiple pairs of such replicas, pick R_j and R_h with the smallest IDs.



(a) Fast path

(b) Slow path

Figure 2: Communication pattern for optimistic Arena.

This is important for proving Lemma 7: if we ban multiple of such replicas, it is possible to ban more honest replicas than faulty ones. Before sending the new-view message, the new leader needs to wait for 2Δ to gathering the blame and Status messages, which is useful in proving Lemma 4.

If the new-view message is detected to be wrong, the replicas will initialize another view-change. Otherwise, the replicas start processing requests by making the first proposals in this view. They will also forward the new-view message. If the view leader only sends the new-view message to a subset of replicas, the rest can still receive the forwarded new-view message and they will start to propose from the next epoch.

4.2. Safety and liveness

We firstly prove safety.

Lemma 1. *If an honest replica commits a block B_k in view v, then a commit certificate for an equivocating block of B_k will never exist in view v.*

Proof. Suppose an honest replica R_i commits a block B_k at the end of epoch-$(k+1)$ (in view v). Then, it must have voted for B_k in the beginning of epoch-$(k+1)$ (say at time point t). If a commit certificate for an equivocating block $B'_{k'}$ exists, it must be one of two following two cases:

1) At least $f+1$ replicas vote for $B'_{k'}$ at $\geq t+\Delta$. That means at least one honest replica voted for $B'_{k'}$ after seeing R_i's vote, which will not happen.
2) At least $f+1$ replicas vote for $B'_{k'}$ at $\leq t$. In this case, R_i will receive at least one vote for $B'_{k'}$ by $t+\Delta$ and it will not commit B_k.

It is worth mentioning that no honest replica will vote during $(t, t+\Delta)$ as they run in lock-step. □

Lemma 2. *If an honest replica commits a block B_k in view v, then all honest replicas lock on a certified block that is ranked higher than or equal to $C_v(B_k)$ before entering view $v+1$.*

Proof. Suppose an honest replica R_i commits a block B_k at the end of epoch-$(k+1)$ (say at time point t). All replicas will receive $C_v(B_k)$ by $t+\Delta$. It is easy to show that each honest replica R_j will still be in view v at $t+\Delta$: otherwise, due to the conditions for entering a new view, R_j must have sent a blame message at time $\leq t-\Delta$; R_i will receive this blame message by t and it will not commit B_k. Therefore, all honest replicas will receive $C_v(B_k)$ before entering view $v+1$. Furthermore, as we proved in Lemma 1, a certificate for an equivocating block will not exist in view v. Therefore, all honest replicas will lock on a certified block ranked higher than or equal to $C_v(B_k)$ before entering view $v+1$. □

Lemma 3. *If an honest replica commits a block B_k in view v, then any certified block ranked equal to or higher than $C_v(B_k)$ must extend B_k.*

Proof. By Lemma 1, no equivocating certificate exists in view v. Then, any certified block $B'_{k'}$ in view v that is ranked equal to or higher than $C_v(B_k)$ must extend B_k. For larger views, we prove by contradiction.

Let S be a non-empty set of certified blocks ranked higher than $C_v(B_k)$, but do not extend B_k. Let $C_{v^*}(B_{k^*})$ be the block ranked the lowest in S. Notice that if B_{k^*} does not extend B_k, neither B_{k^*-1}. To have $C_{v^*}(B_{k^*})$ exist, at least one honest replica must have received either $\langle \mathsf{propose}, v^*, b_{k^*,j}, \sigma_{k^*,j}, C_{v'}(B_{k^*-1}))\rangle_j$ (with $v' < v^*$) or $\langle \mathsf{propose/vote}, v^*, (b_{k^*+1,j}, \sigma_{k^*+1,j}), (\sigma'_{k^*,j}, \Sigma_{k^*,j}), C_{v^*}(B_{k^*-1})\rangle_j$.

- If it is the former, $C_{v'}(B_{k^*-1})$ must be ranked higher than or equal to $C_v(B_k)$ due to Lemma 2. Then, $C_{v'}(B_{k^*-1})$ should be in S, which contradicts the fact that $C_{v^*}(B_{k^*})$ has the lowest rank in S.
- If it is the later, $C_{v^*}(B_{k^*-1})$ must be ranked higher than $C_v(B_k)$ as $v^* > v$, which leads to the same contradiction as above.

Therefore, the set S is empty. □

Theorem 1 (Safety)**.** *Honest replicas will not commit equivocating blocks at any height.*

Proof. We prove by contradiction. Suppose two equivocating blocks B_k and B'_k are committed at height k. Without loss of generality, we assume $k \leq k'$. Then, by Lemma 3, B'_k extends B_k, which leads to a contradiction. □

Next, we prove liveness.

Lemma 4. *When an honest new leader L' sends the* new-view *message for view $v+1$, all honest replicas are in view $v+1$ and L' has received the* Status *messages from all other honest replicas.*

Proof. Suppose an honest new leader L' sends the new-view message for view $v+1$ at time t. It must have

Pipelined Arena

Steady state. While in view v, a replica R_i runs as follows in the beginning of each Δ-epoch:

1) **Epoch-k.** R_i proposes for height-k, votes for height-$(k-1)$, and commits for height-$(k-2)$.
   a) **Propose.** R_i constructs a batch $b_{k,i}$ of requests and computes $\sigma_{k,i} := Sig(sk_i, H(b_{k,i})||v||k)$.
   b) **Vote.**
      - Let $\{(b_{k-1,1}, \sigma_{k-1,1}), ..., (b_{k-1,n}, \sigma_{k-1,n})\}$ be the proposals received by R_i in epoch-$(k-1)$. Notice that $(b_{k-1,j}, \sigma_{k-1,j}) = (\perp, \perp)$ if R_i did not receive a proposal from R_j in epoch-$(k-1)$.
      - Let $\sigma'_{k-1,i} := Sig(sk_i, \sigma_{k-1,1}||...||\sigma_{k-1,n})$ and $\Sigma_{k-1,i} := \{\sigma_{k-1,1}, ..., \sigma_{k-1,n}\}$.
   c) **Commit.**
      - Let $\{(b_{k-2,1}, \sigma_{k-2,1}), ..., (b_{k-2,n}, \sigma_{k-2,n})\}$ be the proposals received by R_i in epoch-$(k-2)$. Again, $(b_{k-2,j}, \sigma_{k-2,j}) = (\perp, \perp)$ if R_i did not receive a proposal from R_j.
      - Let $\{\sigma'_{k-2,1}, ..., \sigma'_{k-2,n}\}$ be the signatures in the votes received R_i in epoch-$(k-1)$. Again, $\sigma'_{k-2,j} = \perp$ if R_i did not receive a vote from R_j. Notice that the number of non-$\perp$ signatures is at least $f+1$.
      - If the number of non-$\perp$ signatures in $\{\sigma'_{k-2,1}, ..., \sigma'_{k-2,n}\}$ is $\geq (f+1)$ and they are identical, R_i constructs $B_{k-2} := (b_{k-2,1}||...||b_{k-2,n}, H(B_{k-3}))$. Notice that a request will be removed from B_{k-2} if it conflicts with another request in front of it. The commit certificate $C_v^{f+1}(B_{k-2})$ is computed as an aggregate signature of $\{\sigma'_{k-2,1}, ..., \sigma'_{k-2,n}\}$.

   R_i broadcasts $\left\langle \mathsf{propose/vote}, v, (b_{k,i}, \sigma_{k,i}), (\sigma'_{k-1,i}, \Sigma_{k-1,i}), C_v^{f+1}(B_{k-2}) \right\rangle_i$.
2) **Blame and quit view.**
   a) *Quit view on detecting equivocation.* If R_j is detected for equivocation, R_i constructs $\left\langle \mathsf{blame}, m_{k,j}, m'_{k,j} \right\rangle_i$, where $m_{k,j}$ and $m'_{k,j}$ are two equivocating messages (propose/vote, blame, new-view) sent by R_j.
   b) *Quit view on detecting missing proposals.* If R_i receives two votes $v_{k,h}$ and $v_{k,j}$ that contain different Σs, it constructs $\langle \mathsf{blame}, v_{k,j}, v_{k,h} \rangle_i$.

   In either case, it stops committing any blocks in the current epoch and broadcasts the blame message in the beginning of the next epoch.

View change. Let L and L' be the leaders of view v and $v+1$ respectively for coordinating the view-change. R_i runs the following steps to switch from view v to $v+1$.

1) **Status.** Upon receiving a valid blame message, R_i stops committing any blocks in the current epoch and forwards it to all other replicas in the beginning of the next epoch. Then, it waits for 2Δ, during which it keeps sending blames when receiving messages that can trigger "Blame and quit view". Upon timeout, it locks on $C_{v'}(B_{k'})$, where $B_{k'}$ is the highest certified block known to it. Then, it sends a Status message that contains $C_{v'}(B_{k'})$ to the new leader L', and enters view $v+1$.
2) **New-view.** L' waits for 2Δ after entering view $v+1$. Then, it broadcasts $\langle \mathsf{new\text{-}view}, v+1, \{\mathsf{R}\}, C_{v'}(B_{k'}) \rangle_{L'}$, where $C_{v'}(B_{k'})$ is a highest certified block known to L', and $\{\mathsf{R}\}$ is a set of replicas to be banned to propose blocks:
   - For "Quit view on detecting equivocation", the equivocation sender R_j will be banned.
   - For "Quit view on detecting missing proposals", the sender R_j of the proposal and the sender R_h of the vote (that does not contain that proposal) will be banned. If there are multiple pairs of such replicas, pick R_j and R_h with the smallest IDs.
3) **First propose.** Upon receiving $\langle \mathsf{new\text{-}view}, v+1, \{\mathsf{R}\}, C_{v'}(B_{k'}) \rangle_{L'}$,
   - if $C_{v'}(B_{k'})$ has a rank lower than R_i's locked block, or $\{\mathsf{R}\}$ includes a pair $(\mathsf{R}_j, \mathsf{R}_h)$ with IDs higher than the pair in the blame sent by R_i, R_i initializes view-change again by broadcasting a blame;
   - otherwise, R_i broadcasts $\langle \mathsf{new\text{-}view}, v+1, \{\mathsf{R}\}, C_{v'}(B_{k'}) \rangle_{L'}$ and $\langle \mathsf{propose}, v+1, b_{k'+1,i}, C_{v'}(B_{k'}) \rangle_i$ in the beginning of the next epoch.

Figure 3: Pipelined Arena.

forwarded a valid blame message at time $t - 4\Delta$. Then, other honest replicas must have received this blame message by $t - 3\Delta$; they will enter view $v + 1$ and send the Status messages by time $t - \Delta$. L' must have received these Status messages by t. □

Lemma 5. *Assume all replicas are initialized (and start to propose) at the same time. Then, they will always run in lock-step: start and end every Δ-epoch at the same time.*

Proof. In a steady state, it is clear that all replicas run in lock-steps. When view-change happens,

1) If L' is honest, by Lemma 4, it will run with other replicas in lock-steps;
2) If L' faulty, it will trigger another view-change, until case 1 happens.

□

Lemma 6. *All honest replicas ban the same replicas.*

Proof. Suppose the view-leader-of-the-next-view L' receives a blame in epoch e (or it receives a message that can trigger "blame and quit view" in epoch e), it stops committing any blocks in the current epoch, and forwards the blame to all other replicas in the beginning of the epoch $e + 1$ (say at time t).

Due to synchrony, all replicas will receive this blame by $t + \Delta$. They will wait for 2Δ before entering the new-view, during which they keep sending and forwarding blames. They will enter the new-view at $t + 3\Delta$ (L' enters the new-view at $t + 2\Delta$).

After entering the new-view, L' waits for another 2Δ and then broadcasts the new-view message. Namely, it broadcasts the new-view message at $t + 4\Delta$. As long as a replica has sent a blame to one honest replica by $t+3\Delta$, this message will be forwarded and received by L' by $t + 4\Delta$.

As other replicas stop sending/forwarding blames at $t + 3\Delta$, L' is guaranteed to receive *all* blames by $t + 4\Delta$. That is to say, L' will hold a complete set of blame messages.

Of course, a faulty L' can equivocate in the new-view message (sending different banned replicas to different subsets of replicas). However, this new-view message will be forwarded by replicas. Therefore, replicas can detect the equivocation by L' and trigger another view-change. A faulty L' can also send a blame that is not the one with the smallest IDs, which can also trigger a view-change. □

Lemma 7. *Honest majority always holds among the replicas that can make proposals.*

Proof. We assume that, after initialization, f of the $(2f+1)$ replicas are faulty, which satisfies honest majority. However, after a view-change, some replicas will be banned to make proposals. By Lemma 6, all honest replicas ban the same replicas. Now, we need to prove that the number of banned honest replicas is no more than that of faulty replicas. Recall that there are two kinds of blame messages:

1) *Quit view on detecting equivocation.* Notice that an honest replica will neither propose nor vote for equivocating proposals. Therefore, this blame will never ban an honest replica.
2) *Quit view on detecting an inconsistency between a proposal and a vote.* Notice that when an honest replica proposes b_i in the beginning of an epoch, this proposal is guaranteed to be received by all other replicas at the end of the epoch. As replicas always run in lock-steps (by Lemma 5), b_i will be included in all honest replicas' votes. Consequently, this blame happens only when a faulty replica sends a proposal to a subset of replicas, or a faulty replica falsely claims that it did not receive a proposal from another replica. Therefore, this blame will ban two replicas and at most one of them is honest.

As a result, honest majority also holds for the rest of replicas after view-change. □

Theorem 2 (Liveness)**.** *A legitimate request will be committed eventually.*

Proof. Recall that faulty replicas can prevent progress through two strategies: stalling and equivocating.

- If they stall and do not make proposals, there will always be honest replicas left (by Lemma 7) to keep the protocol running.
- If they equivocate, view-change will be triggered and at least one replica will be banned. Therefore, view-change can be triggered for at most f times.

In summary, a legitimate request will eventually be committed in a steady state. □

5. Optimistic Arena in Detail

In this section, we provide a detailed description and correctness proof for optimistic Arena.

5.1. The protocol

We present the details of optimistic Arena in Figure 4.

Optimistic Arena only requires replicas to be initialized (and start to propose) within a time interval of Δ. As we mentioned in Section 3.2, we rely on the commit certificate to limit the discrepancies among honest replicas to Δ for all time. That is, a replica R_i can propose for height-k only when it has received a certificate for height-$(k-1)$ or it has received enough information to form a certificate for height-$(k-1)$.

The propose message is the same as that in pipelined Arena: it includes a batch of requests together with a signature on the batch. The vote message in optimistic Arena is different from that in pipelined Arena. It does not need to include the signatures, because replicas can detect misbehaviours based on the ack messages: if faulty replicas equivocate, honest replicas will receive different acks.

It is worth mentioning that, due to the Δ discrepancy, it is possible for a replica to receive $(f + 1)$ consistent votes before receiving the conflict acks. In this case, a block certificate will be formed and broadcasted by this replica; and other replicas will receive and follow this block

Optimistic Arena

Steady state. While in view v, a replica R_i runs the following steps in iterations:

1) **Propose.** Upon receiving $C_v(B_{k-1})$, R_i constructs a batch $b_{k,i}$ of requests, computes $\sigma_{k,i} := Sig(sk_i, H(b_{k,i})||v||k)$, and broadcasts $\langle \mathsf{propose}, v, b_{k,i}, \sigma_{k,i}, C_v(B_{k-1})\rangle$. Meanwhile, it sets $\mathsf{propose\text{-}timer}_{k,i}$ to 2Δ and starts counting down.
2) **Ack.** Let $\{(b_{k,1}, \sigma_{k,1}), ..., (b_{k,n}, \sigma_{k,n})\}$ be the proposals received by R_i. Notice that $(b_{k,j}, \sigma_{k,j}) = (\perp, \perp)$ if R_i did not receive a proposal from R_j. Let $\sigma'_{k,i} := Sig(sk_i, \sigma_{k,1}||...||\sigma_{k,n})$ and $\Sigma_{k,i} := \{\sigma_{k,1}, ..., \sigma_{k,n}\}$. R_i broadcasts $\left\langle \mathsf{ack}, v, \sigma'_{k,i}, \Sigma_{k,i} \right\rangle$ to other replicas using either of the following rules:
   a) *Responsive ack.* Upon receiving $(2f+1)$ proposals, it stops $\mathsf{propose\text{-}timer}_{k,i}$ and broadcasts.
   b) *Synchronous ack.* Upon the timeout of $\mathsf{propose\text{-}timer}_{k,i}$, it broadcasts.

   In either case, R_i sets $\mathsf{vote\text{-}timer}_{k,i}$ to Δ and starts counting down.
3) **Vote.** Upon the timeout of $\mathsf{vote\text{-}timer}_{k,i}$, R_i broadcasts $\langle \mathsf{vote}, v, H(B_k)\rangle_i$, where $B_k := (b_{k,1}||...||b_{k,n}, H(B_{k-1}))$. Notice that a request will be removed from B_k if it conflicts with another request in front of it.
4) **Commit.** R_i commits B_k using either of the following rules:
   a) *Responsive commit.* Upon receiving $(2f+1)$ non-conflicting acks for B_k, it stops $\mathsf{vote\text{-}timer}_{k,i}$ and commits B_k. It computes the commit certificate $C_v^n(B_k)$ as an aggregate signature of $\{\sigma'_{k,1}, ..., \sigma'_{k,n}\}$.
   b) *Synchronous commit.* Upon receiving $(f+1)$ non-conflicting votes for B_k, it commits B_k. It computes the commit certificate $C_v^{f+1}(B_k)$ as an aggregate signature of the $(f+1)$ votes.

   In either case, it needs to broadcast $C_v(B_k)$, but this message is combined with the next proposal.
5) **Blame and quit view.**
   a) *Quit view on detecting equivocation.* If R_j is detected for equivocation, R_i broadcasts $\left\langle \mathsf{blame}, m_{k,j}, m'_{k,j} \right\rangle_i$, where $m_{k,j}$ and $m'_{k,j}$ are two equivocating messages (propose, ack, vote, blame, new-views) sent by R_j.
   b) *Quit view on detecting missing proposals.* If R_i receives two acks $a_{k,h}$ and $a_{k,j}$ that contain different Σs, it broadcasts $\langle \mathsf{blame}, a_{k,j}, a_{k,h}\rangle_i$.

   In either case, it aborts all view v timers and quit view v.

View change. Let L and L' be the leaders of view v and $v+1$ respectively for coordinating the view-change. R_i runs the following steps to switch from view v to $v+1$.

1) **Status.** Upon receiving a valid blame message, R_i stops committing any blocks and forwards it to all other replicas. Then, it waits for 2Δ, during which it keeps sending blames when receiving messages that can trigger "Blame and quit view". Upon timeout, it locks on $C_{v'}(B_{k'})$, where $B_{k'}$ is the highest certified block known to it. Then, it sends a Status message that contains $C_{v'}(B_{k'})$ to the new leader L', and enters view $v+1$.
2) **New-view.** L' waits for 2Δ after entering view $v+1$. Then, it broadcasts $\langle \mathsf{new\text{-}view}, v+1, \{\mathsf{R}\}, C_{v'}(B_{k'})\rangle_{L'}$, where $C_{v'}(B_{k'})$ is a highest certified block known to L', and $\{\mathsf{R}\}$ is a set of replicas to be banned to propose blocks:
   - For "Quit view on detecting equivocation", the equivocation sender R_j will be banned.
   - For "Quit view on detecting missing proposals", the sender R_j of the proposal and the sender R_h of the ack will be banned. If there are multiple pairs of such replicas, pick R_j and R_h with the smallest IDs.
3) **First propose.** Upon receiving $\langle \mathsf{new\text{-}view}, v+1, \{\mathsf{R}\}, C_{v'}(B_{k'})\rangle_{L'}$,
   - if $C_{v'}(B_{k'})$ has a rank lower than R_i's locked block, or $\{\mathsf{R}\}$ includes a pair $(\mathsf{R}_j, \mathsf{R}_h)$ with IDs higher than the pair in the blame sent by R_i, R_i initializes view-change again by broadcasting a blame;
   - otherwise, R_i broadcasts $\langle \mathsf{new\text{-}view}, v+1, \{\mathsf{R}\}, C_{v'}(B_{k'})\rangle_{L'}$ and $\langle \mathsf{propose}, v+1, b_{k'+1,i}, C_{v'}(B_{k'}))\rangle_i$.

Figure 4: Optimistic Arena.

certificate before entering the next view (we prove this in Lemma 9).

The fast path of optimistic Arena requires only 2δ to commit a block. We remark that this is likely to be the common case, as replicas rarely experience crashes or faults, and network connections tend to be reliable. We could also include the proposals in the ack messages to increase the possibility of the fast path being taken. However, this incurs more communication overhead for broadcasting the ack messages. This design choice can be determined based on the quality of the network connection.

The blame mechanism and the view-change protocol in optimistic Arena are similar to those in pipelined Arena.

5.2. Safety and liveness

Lemma 8. *If an honest replica commits a block B_k in view v, then a commit certificate for an equivocating block of B_k will never exist in view v.*

Proof. We first consider responsive commit. If an honest replica R_i commits a block B_k in view v using the responsive commit rule, it must have received $2f+1$ acks, i.e., $C_v^n(B_k)$, which means *all* replicas have sent the same Σ. On the other hand, if a certificate (either responsive or synchronous) for an equivocating block $B'_{k'}$ exists in view v, there must be at least one honest replica that sent ack for $B'_{k'}$ in view v. This is a contradiction because an honest replica will never send acks for equivocating blocks.

Next, we consider that an honest replica R_i commits B_k in view v using the synchronous commit rule. Then, it will never send an ack for an equivocating block $B'_{k'}$ in the same view. Consequently, a responsive certificate for $B'_{k'}$ will never exits in view v, as it requires $2f+1$ acks. To this end, we only need to consider synchronous certificates for $B'_{k'}$. Suppose R_i commits B_k at time t, there must be at least one honest replica R_j that has sent a vote for B_k at time $\leq t$ (sent an ack for B_k at time $\leq t-\Delta$), Then, all replicas must have received R_j's ack by time t.

- No honest replica will send an ack or a vote for an equivocating block $B'_{k'}$ at time $\geq t$.
- No honest replica will send an ack for an equivocating block $B'_{k'}$ at time $< t-\Delta$ (otherwise R_i will receive the ack and will not commit B_k). Consequently, no honest replica will send a vote for $B'_{k'}$ at time $< t$.

This implies that no honest replica will vote for $B'_{k'}$ in view v, hence a synchronous certificate for $B'_{k'}$ that requires $(f+1)$ votes will not exit in view v. □

Lemma 9. *If an honest replica commits a block B_k in view v, then all honest replicas lock on a certified block that is ranked higher than or equal to $C_v(B_k)$ before entering view $v+1$.*

Proof. Suppose an honest replica R_i commits a block B_k at time point t. All replicas will receive $C_v(B_k)$ by $t+\Delta$. It is easy to show that each honest replica R_j will still be in view v at $t+\Delta$: otherwise, due to the conditions for entering a new view, R_j must have sent a blame message at time $\leq t-\Delta$; R_i will receive this blame message by t and it will not commit B_k. Therefore, all honest replicas will receive $C_v(B_k)$ before entering view $v+1$. Furthermore, as we proved in Lemma 8, a certificate for an equivocating block will not exist in view v. Therefore, all honest replicas will lock on a certified block ranked higher than or equal to $C_v(B_k)$ before entering view $v+1$. □

Lemma 10. *If an honest replica commits a block B_k in view v, then any certified block ranked equal to or higher than $C_v(B_k)$ must extend B_k.*

The proof is similar to that of Lemma 3 except that Lemma 8 and 9 need to be invoked.

Proof. By Lemma 8, no equivocating certificate exists in view v. Then, any certified block $B'_{k'}$ in view v that is ranked equal to or higher than $C_v(B_k)$ must extend B_k. For larger views, we prove by contradiction.

Let S be a non-empty set of certified blocks ranked higher than $C_v(B_k)$, but do not extend B_k. Let $C_{v^*}(B_{k^*})$ be the block ranked the lowest in S. Notice that if B_{k^*} does not extend B_k, neither B_{k^*-1}. To have $C_{v^*}(B_{k^*})$ exist, at least one honest replica must have received $\langle \mathsf{propose}, v^*, b_{k^*,j}, \sigma_{k^*,j}, C_{v'}(B_{k^*-1}))\rangle_j$.

- If $v' < v^*$, $C_{v'}(B_{k^*-1})$ must be ranked higher than or equal to $C_v(B_k)$ due to Lemma 9. Then, $C_{v'}(B_{k^*-1})$ should be in S, which contradicts the fact that $C_{v^*}(B_{k^*})$ has the lowest rank in S.
- If $v' = v^*$, $C_{v'}(B_{k^*-1})$ must be ranked higher than $C_v(B_k)$ due to the fact that $v^* > v$, which leads to the same contradiction as above.

Therefore, the set S is empty. □

Theorem 3 (Safety)**.** *Honest replicas will not commit equivocating blocks at any height.*

Proof. We prove by contradiction. Suppose two equivocating blocks B_k and B'_k are committed at height k. Without loss of generality, we assume $k \leq k'$. Then, by Lemma 10, B'_k extends B_k, which leads to a contradiction. □

Next, we prove liveness.

Lemma 11. *Assume all replicas are initialized (and start to propose) within a time interval of Δ. Then, the time interval for all honest replicas to propose blocks is always Δ.*

Proof. Given that all replicas are initialized (and start to propose) within a time interval of Δ, replicas are guaranteed to receive all proposals sent from honest replicas within 2Δ. Then, there are two cases:

1) *View-change does not happen.* Whenever an honest replica commits a block and proposes the next block, it will include the commit certificate in its proposal. Other honest replicas, receiving the commit certificate within Δ, will propose their blocks immediately as well.
2) *View-change happens.* Suppose an honest replica R_i receives a blame and forwards it to others at t.
   - If the sender of the blame is honest, it must have sent this blame to all replicas at time t' with $t-$

$\Delta < t' < t$. Then, all other honest replicas must receive this blame at time between t' and $t' + \Delta$, with $t' < t < t' + \Delta$.

- If the sender of the blame is faulty and it sends the blame to a subset of replicas (w.l.o.g., R_i is the first receiver), all honest replicas will still receive the forwarded blame from R_i by $t + \Delta$.

In either case, all honest replicas will enter the new view and make the next proposal within a time interval of Δ.

□

Lemma 12. *All honest replicas ban the same replicas.*

Proof. The proof of this lemma is similar to that of Lemma 6. Replicas are no longer run in an epoch style. Suppose the view-leader-of-the-next-view L' receives a blame at t (or it receives a message that can trigger "blame and quit view" at t), it stops committing any blocks and forwards the blame to all other replicas. Then, the remaining proof is the same as that of Lemma 6. □

Lemma 13. *Honest majority always holds among the replicas that can propose blocks.*

Proof. We assume that, after initialization, f of the $(2f+1)$ replicas are faulty, which satisfies honest majority. However, after a view-change, some replicas will be banned to make proposals. By Lemma 12, all honest replicas ban the same replicas. Now, we need to prove that the number of banned honest replicas is no more than that of faulty replicas. Recall that there are two kinds of blame messages:

1) *Quit view on detecting equivocation.* Notice that an honest replica will never propose, ack or vote for equivocating proposals. Therefore, this blame will never ban an honest replica.
2) *Quit view on detecting an inconsistency between a proposal and an ack.* Notice that when an honest replica proposes b_i, this proposal is guaranteed to be received by all other replicas (by Lemma 11 and the fact that honest replicas will wait for 2Δ after sending a proposal). That means b_i will be included in all honest replicas' acks. Consequently, this blame happens only when a faulty replica sends a proposal to a subset of replicas, or a faulty replica falsely claims that it did not receive a proposal from another replica. Therefore, this blame will ban two replicas and at most one of them is honest.

As a result, honest majority also holds for the rest of replicas after a view-change. □

Theorem 4 (Liveness)**.** *A legitimate request will be committed eventually.*

Proof. Recall that faulty replicas can prevent progress through two strategies: stalling and equivocating.

- If they stall and do not make proposals, there will always be honest replicas left (by Lemma 13) to keep the protocol running.
- If they equivocate, view-change will be triggered and at least one replica will be banned. Therefore, view-change can be triggered for at most f times.

In summary, a legitimate request will eventually be committed in a steady state. □

6. Implementation and Evaluation

In this section, we systematically evaluate Arena and compare it with the state-of-the-art synchronous BFT-SMRs: Sync HotStuff [2] and the fast path of OptSync [4][1]. We remark that our experimental setup closely follows OptSync [4]; we aim to provide a head-to-head comparison with OptSync, as it is our closest competitor.

6.1. Implementation

We implemented Arena, based on the open-source C++ implementation of Sync HotStuff[2]. It is worth mentioning that the open-source code of OptSync is also based on this implementation. Therefore, all three BFT-SMRs to be evaluated have the same code base.

Each replica runs on a separate AWS VM with eight 3.3GHz vCPUs and 16 GB RAM, running Ubuntu Server 18.04.1 LTS 64. The maximum network bandwidth is 5Gbps. We generate open-loop clients on a separate VM with the same configuration. The RTT (which can be considered as δ) between any two VMs is 0.1ms, and we set Δ as 300ms[3].

Following the setting of [4], we batch 400 requests in a proposal for all three BFT-SMRs. We consider two types of request size: 8-byte and 1KB, which correspond to the 0/0 payload and 1024/1024 payload in [4]. Also, following [4], we only measure the fast path performance (where no faulty replica exists), as the slow path performance of such protocols solely depends on Δ.

We repeat all experiments five times and report the average values with error bars indicating standard deviations.

6.2. Throughput vs. latency

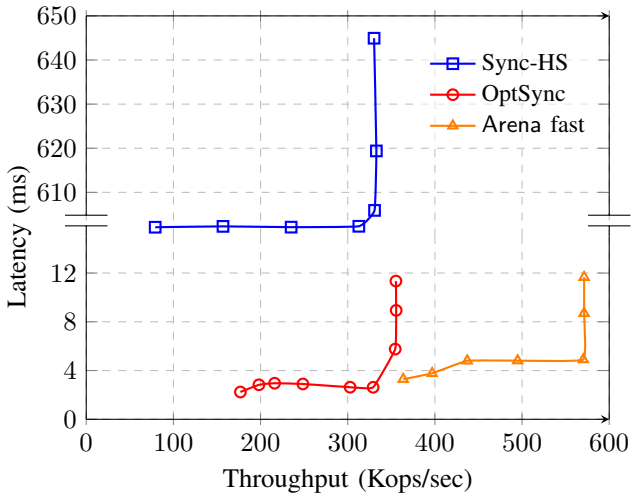
We fix the number of replicas (n) as three, increase the number of concurrent clients, and measure throughput vs. latency. At first, when more clients are added, the throughput increases while the latency stays the same. However, once the system becomes saturated, the throughput remains constant, causing the latency to increase with additional clients.

Figure 5a shows the results for the case where the request size is small (8-byte). The fast path of optimistic Arena can reach a peak throughput of 570 261 TPS, with a latency of around 4.9ms It has a comparable stable latency to

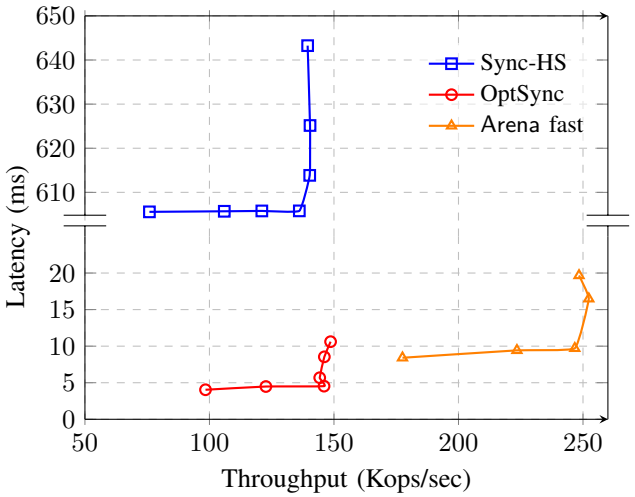
1. There are two versions of OptSync and their fast paths have the same performance.

2. https://github.com/hot-stuff/librightstuff

3. A common way for reducing fragility of synchronous protocol is setting a large Δ, and our protocol can still have a good performance due to the optimistic responsiveness.

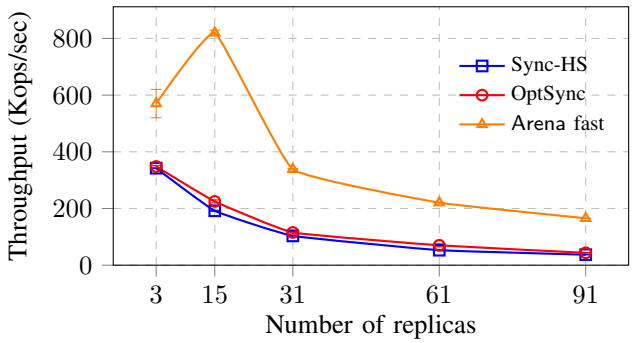


(a) Throughput vs. Latency with request size of 8-byte.

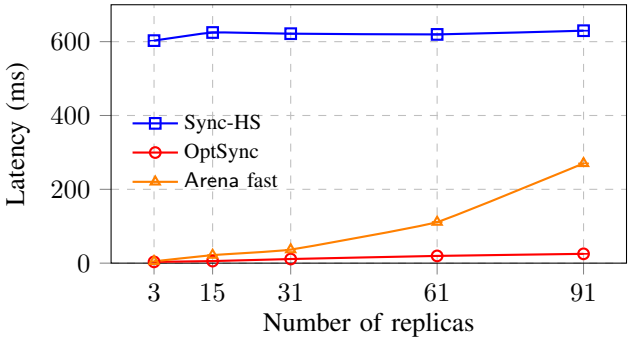


(b) Throughput vs. Latency with request size of 1KB.

Figure 5: Throughput vs. Latency ($n = 3$, $f = 1$, $\Delta = 300$ms).



(a) Throughput.



(b) Latency.

Figure 6: Scalability (request size of 8-byte).

OptSync, but with 1.6× higher peak throughput. Its advantage is more prominent when compared with Sync HotStuff: 20.8× lower latency and 1.7× higher throughput.

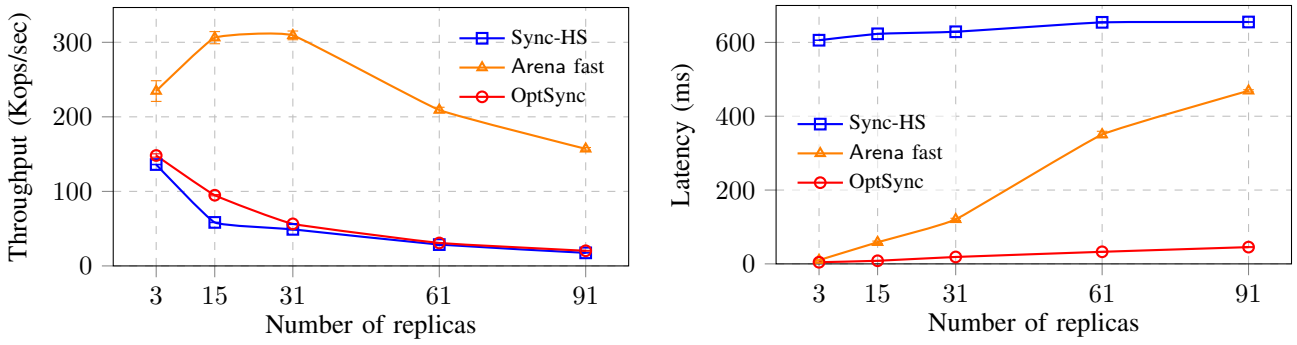
Figure 5b shows the results for processing 1KB requests. The throughput becomes lower for all systems due to the larger bandwidth consumption for each request. However, the peak throughput of optimistic Arena (fast path) is still around 1.8× higher than that of Sync HotStuff and 1.7× higher than that of OptSync. It can reach a peak throughput 246 703 TPS, with a latency of around 9.71ms.

6.3. Scalability

Next, we measure the scalability by increasing the number of replicas (n). We measure the throughput and latency when systems become saturated. Figure 6 and 7 show the results.

Figure 6a and Figure 7a present the throughput of these systems for 8-byte and 1KB requests respectively. Sync HotStuff and OptSync suffer significant throughput drops with more replicas, because more replicas incur more communication for these systems. This is also the case for Arena, but all replicas in Arena can propose requests, making its throughput higher. When $n = 15$, optimistic Arena (fast path) achieves a throughput (819 708 TPS) around 4.3× higher than Sync HotStuff, and 3.6× higher than OptSync for 8-byte requests; it achieves a throughput (306 183 TPS) 5.2× higher than Sync HotStuff and 3.2× higher than OptSync for 1KB requests. As the number of replicas increases, Arena becomes more advantageous. Specifically, when $n = 91$, optimistic Arena (fast path) achieves a throughput (1 656 TPS) around 4.4× higher than Sync HotStuff, 3.7× higher than OptSync for 8-byte requests; it achieves a throughput (157.079 TPS) 8.9× than Sync HotStuff and 7.7× than OptSync for 1KB requests.

When considering latency, optimistic Arena (fast path) and OptSync is superior due to their responsive commit. OptSync scales better than Arena, because OptSync requires fewer votes to trigger responsive commit. When n =15, optimistic Arena (fast path) is 28.6× faster than Sync HotStuff and 3.8× slower than OptSync for 8-byte requests; and it is 10.6× faster than Sync HotStuff and 6.9× slower than



(a) Throughput.

(b) Latency.

Figure 7: Scalability (request size of 1KB).

OptSync for 1KB requests. When $n = 91$, optimistic Arena (fast path) is 2.3× faster than Sync HotStuff and 10.7× slower than OptSync for 8-byte requests; and it is 1.4× faster than Sync HotStuff and 10.3× slower than OptSync for 1KB requests.

7. Related Work

7.1. Synchronous BFT-SMR

Dfinity [9] is a synchronous BFT-SMR that runs in lock-steps. In the beginning of every epoch, a leader proposes a block and waits for 2Δ, during which replicas vote for the best ranked block(s) they received. Namely, a replica forwards its voted block to all other replicas; if a replica receives a block with a rank equal to or better than the best ranked block it has voted so far, it votes for that block. This process continues until a certificate is formed, after which they enter the next epoch. At first glance, Dfinity follows a multi-leader design. However, unlike multi-leader SMR, Dfinity can only commit one proposal in each consensus round.

Guo et al. [12] introduce the mobile sluggish model, which allows some replicas to be sluggish, i.e., messages sent/receive by sluggish replicas can be more than Δ. PiLi [10] is a BFT-SMR that runs in the mobile sluggish model. It also executes in lock-steps, where each epoch lasts for 5Δ. It commits five blocks for every 13 consecutive epochs.

Sync HotStuff [2] gets rid of the lock-step execution based on a very simple and intuitive design. In more detail, it introduce a leader for proposing blocks (like PBFT [13] or Paxos [14]); each replica can commit after waiting for the maximum round-tip delay (i.e., 2Δ) unless it receives by that time an equivocating block signed by the leader. If the leader does not propose until a timeout, replicas run a view-change to perform a leader change. Abraham et al. [3] further reduce the commit latency from 2Δ to $\Delta + 2\delta$.

Kiayias et al. presented Ouroboros [15], a proof-of-stake protocol with an innovative reward mechanism that guarantees honest behavior as an approximate Nash equilibrium, neutralizing attacks such as selfish mining. Bagaria et al. presented Prism [16], a new proof-of-work blockchain protocol, achieving performance scalable by total decoupling of transaction proposing, validation, and confirmation functionalities. Pass et al. presented FruitChains [17], a proof-of-work protocol that reduces the variance of mining rewards, thus significantly reducing or even eliminating the need for mining pools.

7.2. Optimistic responsiveness

The notion of optimistic responsiveness was firstly introduced in Thunderella [11], which makes the observation that it is safe to commit a block in $O(\delta)$ if $> 3n/4$ votes have been received. If the responsive commit (i.e., fast path) cannot be made, an explicit switch is performed to move to a Nakamoto-style consensus (i.e., slow path). This explicit switch between the two paths incurs an undesired latency.

Shrestha et al. [4] propose two BFT-SMRs with optimistic responsiveness, but do not require any explicit switch between the fast path and slow path, i.e., replicas exist in both paths simultaneously. Our optimistic Arena follows this paradigm with a higher throughput.

7.3. Multi-leader SMR

To improve the performance of leader-based SMRs, multi-leader SMRs have been proposed. For example, in Mencius [18], every replica acts as a leader for the sequence numbers assigned to it. For example, R_i is a leader for all sequence numbers j that satisfies $(j \mod n = i)$. Ideally, Mencius can achieve a throughput that is n times larger than leader-based SMRs. However, once a crash occurs, the throughput of Mencius will quickly drop to zero until a revocation starts that makes all correct replicas learn of no-ops for instances coordinated by the faulty replica. EPaxos [19] introduces a dependency graph to track the relationship of different requests. Benefits from this approach, non-conflicting requests can be committed in a fast path of two message delays. However, replicas need to spend more time for checking the dependencies; and in the presence of conflicts, it takes a slow path of four message delays.

M^2Paxos [20] gets rid of the dependency graph by assigning different objects to different replicas and enforcing requests accessing the same objects to be ordered by the same replica. However, a request may access objects maintained by different replicas, hence M^2Paxos still needs to resolve conflicts. Based on the observation that concurrent failures in geo-distributed systems are rare, Enes et al. presented Atlas [21], which is a multi-leader SMR trading off fault tolerance for scalability. The size of the fast quorum Q depends on f: $Q = \lfloor n/2 \rfloor + f$. Avarikioti et al. presented FnF-BFT [22], a multi-leader SMR protocol designed for the partially synchronous model that utilizes historical data to ensure that well-performing replicas are in command.

7.4. DAG

Keidar et al. [23] propose a DAG-based protocol named DAG-Rider using reliable broadcast as a basic building block to construct a DAG. Replicas order the proposals based on the local DAG without communication. However, DAG-Rider takes in expectation six rounds of reliable broadcast, resulting in a long tail latency. Tusk [24] and Bullshark [25] exploit synchronous periods to improve the rounds of reliable broadcast to five and two, respectively.

7.5. Asynchronous BFT

Ben-Or's randomized binary consensus [26] is the main component for constructing a leaderless SMR. Ezhilchelvan et al. [27] use a common coin [28] to reduce the average number of message delays and allow proposers to propose arbitrary values. Pedone et al. [29] exploit the weakly ordering guarantees from the network layer. Cachin et al. [30] propose a Diffie-Hellman based coin-tossing protocol and construct a practical and theoretically optimal Byzantine agreement protocol. Its efficiency and robustness were further improved in [31], [32], [33]. Lu et al. [34] presented Dumbo-MVBA, which leverages asynchronous provable dispersal broadcast (APDB), allowing each input to be split and dispersed to every party and later recovered efficiently. Yang et al. [35] presented DispersedLedger, designed for variable network bandwidth, which enables replicas to propose, order, and agree on blocks without downloading the full content by decoupling the bandwidth-intensive block downloads across different replicas. Nazirkhanova et al. [36] presented Semi-AVID-PR, a storage- and communication-efficient protocol that utilizes linear erasure-correcting codes and homomorphic vector commitments and does not require comprehensive termination properties. Cohen et al. [37] investigated the efficiency of the retrieve sub-protocol in AVID (Asynchronous Verifiable Information Dispersal) and achieved complete retrieval with expected $O(\log n)$ messages per node using a probabilistic retrieval algorithm.

8. Conclusion

In this paper, we introduce a completely novel direction for running synchronous SMRs, dubbed Arena. Compared with other partially synchronous multi-leader SMR, Arena has a much simpler design, thanks to the synchrony assumption. Furthermore, Arena is more robust: "no progress" of a leader will not trigger a view-change. Arena significantly outperforms the state-of-the-art in head-to-head comparisons. We fully implement Arena and systematically evaluate its performance. Our experimental results show that Arena achieves a peak throughput up to 8.9× higher than Sync HotStuff and 7.7× higher than OptSync.

References

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

[2] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin, "Sync hotstuff: Simple and practical synchronous state machine replication," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 106–118.

[3] I. Abraham, K. Nayak, L. Ren, and Z. Xiang, "Byzantine agreement, broadcast and state machine replication with near-optimal good-case latency," *arXiv preprint arXiv:2003.13155*, 2020.

[4] N. Shrestha, I. Abraham, L. Ren, and K. Nayak, "On the optimality of optimistic responsiveness," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 839–857.

[5] S. Gupta, J. Hellings, and M. Sadoghi, "Rcc: Resilient concurrent consensus for high-throughput secure transaction processing," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 1392–1403.

[6] C. Stathakopoulou, T. David, and M. Vukolic, "Mir-bft: High-throughput bft for blockchains," *arXiv preprint arXiv:1906.05552*, 2019.

[7] M. Eischer and T. Distler, "Egalitarian byzantine fault tolerance," in *2021 IEEE 26th Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, 2021, pp. 1–10.

[8] C. Stathakopoulou, M. Pavlovic, and M. Vukolić, "State machine replication scalability made simple," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 17–33.

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

[10] T. H. Chan, R. Pass, and E. Shi, "Pili: An extremely simple synchronous blockchain," *Cryptology ePrint Archive*, 2018.

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

[12] Y. Guo, R. Pass, and E. Shi, "Synchronous, with a chance of partition tolerance," in *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part I 39*. Springer, 2019, pp. 499–529.

[13] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99. USA: USENIX Association, 1999, p. 173–186.

[14] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007*, I. Gupta and R. Wattenhofer, Eds. ACM, 2007, pp. 398–407. [Online]. Available: https://doi.org/10.1145/1281100.1281103

[15] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Annual international cryptology conference*. Springer, 2017, pp. 357–388.

[16] V. Bagaria, S. Kannan, D. Tse, G. Fanti, and P. Viswanath, "Prism: Deconstructing the blockchain to approach physical limits," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 585–602.

[17] R. Pass and E. Shi, "Fruitchains: A fair blockchain," in *Proceedings of the ACM symposium on principles of distributed computing*, 2017, pp. 315–324.

[18] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building efficient replicated state machine for wans," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 369–384. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/mao/mao.pdf

[19] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, M. Kaminsky and M. Dahlin, Eds. ACM, 2013, pp. 358–372. [Online]. Available: https://doi.org/10.1145/2517349.2517350

[20] S. Peluso, A. Turcu, R. Palmieri, G. Losa, and B. Ravindran, "Making fast consensus generally faster," in *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*. IEEE Computer Society, 2016, pp. 156–167. [Online]. Available: https://doi.org/10.1109/DSN.2016.23

[21] V. Enes, C. Baquero, T. F. Rezende, A. Gotsman, M. Perrin, and P. Sutra, "State-machine replication for planet-scale systems," in *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds. ACM, 2020, pp. 24:1–24:15. [Online]. Available: https://doi.org/10.1145/3342195.3387543

[22] Z. Avarikioti, L. Heimbach, R. Schmid, L. Vanbever, R. Wattenhofer, and P. Wintermeyer, "Fnf-bft: Exploring performance limits of bft protocols," *arXiv preprint arXiv:2009.02235*, 2020.

[23] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All you need is dag," in *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, 2021, pp. 165–175.

[24] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and tusk: a dag-based mempool and efficient bft consensus," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 34–50.

[25] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: Dag bft protocols made practical," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2705–2718.

[26] M. Ben-Or, "Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract)," in *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983*, R. L. Probert, N. A. Lynch, and N. Santoro, Eds. ACM, 1983, pp. 27–30. [Online]. Available: https://doi.org/10.1145/800221.806707

[27] P. Ezhilchelvan, A. Mostefaoui, and M. Raynal, "Randomized multivalued consensus," in *Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISORC 2001*. IEEE, 2001, pp. 195–200.

[28] M. O. Rabin, "Randomized byzantine generals," in *24th annual symposium on foundations of computer science (sfcs 1983)*. IEEE, 1983, pp. 403–409.

[29] F. Pedone, A. Schiper, P. Urbán, and D. Cavin, "Solving agreement problems with weak ordering oracles," in *European Dependable Computing Conference*. Springer, 2002, pp. 44–61.

[30] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography," *Journal of Cryptology*, vol. 18, no. 3, pp. 219–246, 2005.

[31] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 31–42.

[32] S. Duan, M. K. Reiter, and H. Zhang, "Beat: Asynchronous bft made practical," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2028–2041.

[33] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, *Dumbo: Faster Asynchronous BFT Protocols*. New York, NY, USA: Association for Computing Machinery, 2020, p. 803–818. [Online]. Available: https://doi.org/10.1145/3372297.3417262

[34] Y. Lu, Z. Lu, Q. Tang, and G. Wang, "Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited," in *Proceedings of the 39th symposium on principles of distributed computing*, 2020, pp. 129–138.

[35] L. Yang, S. J. Park, M. Alizadeh, S. Kannan, and D. Tse, "{DispersedLedger}:{High-Throughput} byzantine consensus on variable bandwidth networks," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 493–512.

[36] K. Nazirkhanova, J. Neu, and D. Tse, "Information dispersal with provable retrievability for rollups," in *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, 2022, pp. 180–197.

[37] S. Cohen, G. Goren, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Proof of availability & retrieval in a modular blockchain architecture," *Cryptology ePrint Archive*, 2022.