Optimal Good-Case Latency for Rotating Leader Synchronous BFT

Ittai Abraham
VMware Research, Herzliya, Israel

Kartik Nayak
Duke University, Durham, NC, USA

Nibesh Shrestha
Rochester Institute of Technology, NY, USA

Abstract

This paper explores the good-case latency of synchronous Byzantine Fault Tolerant (BFT) consensus protocols in the rotating leader setting. We first present a lower bound that relates the latency of a broadcast when the sender is honest and the latency of switching to the next sender. We then present a matching upper bound with a latency of 2Δ (Δ is the pessimistic synchronous delay) with an optimistically responsive change to the next sender. The results imply that both our lower and upper bounds are tight. We implement and evaluate our protocol and show that our protocol obtains similar latency compared to state-of-the-art stable-leader protocol Sync HotStuff while allowing optimistically responsive leader rotation.

2012 ACM Subject Classification Security and privacy → Distributed systems security

Keywords and phrases Distributed Computing, Byzantine Fault Tolerance, Synchrony

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2021.27

Related Version *Previous Version*: `https://eprint.iacr.org/2021/1138.pdf`

1 Introduction

Byzantine fault tolerant (BFT) consensus protocols provide a consistent service despite some malicious and arbitrary process failures. These protocols have been particularly useful in developing infrastructures that span across multiple entities some of which may be incentivized to act maliciously. Thus BFT protocols have been widely adopted to build decentralized blockchain technologies that promise tamper-proof ledger services without a central authority. In accordance, this problem has been studied for over 40 years under various system models and assumptions [26, 2, 7, 22]. Among these, authenticated BFT protocols under *synchrony* assumption are particularly interesting as these protocols can tolerate one-half Byzantine failures [14, 15, 18] compared to partially synchronous or asynchronous protocols which tolerate only upto one-third Byzantine failures [13].

When consensus is to be achieved for a sequence of values, a particularly well-studied approach among BFT protocols is the *stable-leader* approach where a single replica takes lead in coordinating all participating replicas into reaching consensus [7]. Stable-leader BFT protocols usually provide better performance both in terms of throughput and latency as they avoid a *view-change* process to switch leaders which incurs additional cost. From a practical perspective, an important metric in these protocols is the good-case latency to achieve consensus. Informally, good-case latency of a BFT protocol is the time for all honest replicas to commit (over all executions and adversarial strategies), given an honest leader. A recent work [3] provides a complete categorization of the good-case latency of BFT protocols under different network settings and number of faults. In particular, assuming synchrony, they provide matching upper and lower bounds of $\Delta + O(\delta)$ good-case latency where Δ is the known pessimistic network delay and δ is the unknown actual network delay.

While the stable-leader approach is favored for their better performance, many applications require a democracy favoring policy where the participating replicas take turn in being leader to coordinate consensus decisions. We call this a *rotating-leader* approach. The rationale is to obtain better *fairness*, *censorship resistance*, and uniform distribution of work. Indeed, several recent protocols have been designed with this goal in mind [9, 10, 16, 17, 24, 8, 20]. In general, rotating-leader BFT protocols can broadly be classified into two categories. In the first kind, multiple leaders are selected to propose values concurrently in different slots wherein each leader proposes a value in their own instance in the *common log*. While the value proposed by an honest leader can be decided with a small constant latency in the good-case, the value proposed by a Byzantine leader can take a linear number of rounds before it gets finalized in the worst case. In many state machine replication applications, a value proposed in an instance is not useful until all prior instances in the log have been finalized. Thus, such protocols always have worst-case latency which is linear.

The second kind utilizes the "block chaining" paradigm where a leader proposes a value in the form of a *block* which explicitly extends a block B proposed by an earlier leader by including hash of previous block B called its *parent*. In this paradigm, when a block B is committed, all its ancestors are also committed and all blocks in the log up to block B can be safely used. In addition, when the leader of block B is honest, block B and all its ancestors can be committed with a small constant latency in the good-case. This is a natural approach adopted by many existing protocols [26, 2, 8, 17, 25]. In this work, we focus on such rotating-leader BFT protocols that utilize block chaining paradigm. Under this paradigm, existing rotating-leader BFT protocols such as PiLi [10], Dfinity [17, 1] and Streamlet [8] incur a latency of 65Δ, 17Δ and 12Δ respectively to commit a single decision in the good-case. While the good-case latency captures the latency of a commit given an honest leader, a rotating-leader protocol keeps changing leaders through a view-change process each time. Thus, the overall latency of a rotating-leader protocol depends on the combination of good-case latency and the latency of the view-change. In this work, we initiate a study to optimize the combined latency for rotating-leader protocols. A natural approach to optimize the combined latency is to spend as little time during the view-change as possible. Ideally, we want to change leaders and have the new leader propose responsively in $O(\delta)$ time compared to waiting $\Omega(\Delta)$ delay during the view-change process. We refer to such property as *optimistically responsive* rotating leader chained protocol defined as follows:

▶ **Definition 1** (Optimistically Responsive Rotating Leader Chained Protocol, Informal)**.** *We say that a rotating-leader chained protocol has optimistically responsive leader change if for any two consecutive honest leaders p_i and p_{i+1}, p_{i+1} sends its input within $O(\delta)$ time after receiving p_i's input.*

Allowing consecutive leaders to propose responsively allows a protocol to progress at network speed when the leaders are honest. In the context of stable-leader approach, a recent work Sync HotStuff [2] allows a fixed leader to propose responsively. For the rotating-leader approach, few recent works [25, 21] offer solutions that allow consecutive leaders to propose responsively. However, these protocols require a special *optimistic condition* (where $> 3n/4$ replicas are expected to be honest) to provide responsiveness. Moreover, none of the prior works formalize this notion to provide theoretical feasibility results w.r.t. the good-case latency for the rotating-leader approach.

To close this gap, our paper explores the optimal good-case latency for rotating-leader synchronous BFT protocols with responsive proposals. Specifically we ask,

What is the optimal good-case latency of optimistically responsive rotating-leader chained consensus protocol?

We answer this question by providing a lower bound relating the good-case latency for a single slot with that of the view-change. Interestingly, we observe that if the protocol performs view-change in $O(\delta)$, then the commit latency for a single slot cannot be $< 2\Delta$. We also provide a matching upper bound protocol with $2\Delta + O(\delta)$ latency. In essence, our results are tight (ignoring $O(\delta)$ terms). We also evaluate our upper bound protocol against state-of-the art synchronous protocol and observe that our protocol provides similar latency metric compared to Sync HotStuff which is the state-of-the art synchronous consensus protocol that follows stable-leader approach and significantly better compared to its rotating-leader counterpart.

A lower bound on the good-case latency of rotating-leader consensus protocols with responsive proposals. Our first result presents a lower bound on the good-case latency of a rotating-leader consensus protocols with responsive proposals. Specifically, we show the following:

▶ **Theorem 2** (Lower bound on the good-case latency of an optimistically responsive rotating-leader chained consensus protocol, Informal)**.** *There exists an execution in an optimistically responsive rotating-leader chained consensus protocol in an unsynchronized start model tolerating $n/3 \leq f < n/2$ faults where the following two conditions do not hold simultaneously even in executions where messages between honest parties arrive instantaneously and all parties start at time 0: (i) the good-case commit latency is less than 2Δ, and (ii) honest senders propose responsively in $O(\delta)$ time after receiving a proposal from the previous honest sender.*

Intuitively, our lower bound states that if a protocol performs view-change in $O(\delta)$ time, then the commit latency for a single slot has to be at least 2Δ time; otherwise the safety of a commit cannot be guaranteed. This lower bound applies to protocols in an *unsynchronized start model* where parties do not all start the protocol at the same time (explained later).

Optimal responsively proposing rotating-leader protocol with 2Δ-synchronous latency. Our second result presents a matching upper bound protocol that achieves optimal commit latency of 2Δ with responsive proposals. Our upper bound result is presented in the form of state machine replication (SMR) protocol that decides on a sequence of values. In our SMR protocol, the leaders are rotated after each proposal. A sequence of honest leaders can propose within 2δ of previous proposal thus supporting responsive proposals. The good-case commit latency for a single slot is optimal, i.e., 2Δ. Due to responsive leader rotation, our protocol changes leaders before prior proposed values have been committed. For a sequence of k honest leaders, our protocols can commit k values in $2\Delta + O(k\delta)$ time.

Implementation and evaluation. We implement and evaluate the performance of our protocol and compare it with Sync HotStuff [2] which is the state-of-art synchronous protocol with stable-leader approach. In terms of latency, our protocol has similar latency profile compared to Sync HotStuff, i.e., like Sync HotStuff, our protocol commits k values in $2\Delta + O(k\delta)$ time. We validate this in our evaluation by showing a latency comparable to Sync HotStuff. However, Sync HotStuff being a stable-leader protocol does not provide fairness and censorship resistance. To ensure fair comparison, we evaluate our protocol against Sync HotStuff with leader rotation where leaders are rotated after each proposal. Compared to this variation, we obtain *four times better latency* in all configurations.

2 Model and Definitions

We consider a system consisting of n replicas in a reliable, authenticated all-to-all network, where up to $f < n/2$ replicas can be Byzantine faulty. The Byzantine replicas may behave arbitrarily. A replica that is not corrupted is considered to be honest and executes the protocol as specified.

Communication between honest replicas are synchronous. Thus, if a replica r sends a message x to another replica r' at time t, r' receives the message by time $t + \delta$ if r is honest. The delay parameter δ is upper bounded by Δ. The upper bound Δ is known, but δ is unknown to the system. δ can be regarded as an actual delay in the real-world network. We assume all honest replicas have clocks moving at the same speed.

We make use of digital signatures and a public-key infrastructure (PKI) to validate messages and detect equivocation. Message x sent by a node p is digitally signed by p's private key and is denoted by $\langle x \rangle_p$. In addition, we use $H(x)$ to denote the invocation of the random oracle H on input x.

▶ **Definition 3** (Byzantine Fault-tolerant State Machine Replication [23]). *A Byzantine fault-tolerant state machine replication protocol commits client requests as a linearizable log to provide a consistent view of the log akin to a single non-faulty server, providing the following two guarantees. (i)* ***Safety****. Honest replicas do not commit different values at the same log position. (ii)* ***Liveness****. Each client request is eventually committed by all honest replicas.*

▶ **Definition 4** (Good-case Latency [3]). *A Byzantine broadcast (or Byzantine reliable broadcast) protocol has good-case latency of T, if all honest parties commit within time T since the broadcaster starts the protocol (over all executions and adversarial strategies), given the designated broadcaster is honest.*

Chained BFT SMR. As mentioned before, our work focuses on BFT protocols that utilize the block chaining paradigm where a leader proposes a value in the form of a block by explicitly extending a block B_k proposed by an earlier leader by including hash of previous block B_k called its *parent*. A block determines a unique hash chain for all previous blocks in the log. The first block in the chain is called the *genesis* block, and the distance from the genesis block to a block B in the chain is called the *height* of block B. A block B_k at height k has the format, $B_k := (b_k, H(B_{k-1}))$ where b_k denotes the proposed payload at height k, B_{k-1} is the block at height $k-1$ and $H(B_{k-1})$ is the hash digest of B_{k-1}. A block B_k is said to extend a block B_h if $B_k = B_h$ or B_k is a descendant of B_h. All the blocks in the chain from genesis block up to block B_{k-1} are the *ancestors* of block B_k. In this paradigm, when a block B_k is committed, all its ancestors are also committed.

With leaders proposing responsively in $O(\delta)$ time after receiving proposal from a prior leader, an honest leader may propose a new block by extending on a block proposed by a prior Byzantine leader without detecting conflicting block proposals made by the Byzantine leader. When such inconsistencies are detected, a natural solution is to execute some fallback mechanism to collect blocks possibly committed by some honest replicas as the new leader could have proposed blocks extending conflicting proposals. However, such fallback mechanisms worsen the good-case latency of a protocol as it involves notifying of misbehavior by earlier Byzantine leaders and collecting possibly committed blocks to decide on a safe value to extend. In this work, we focus on responsively proposing rotating leader chained BFT SMR protocols that do not involve any fallback mechanism and always decide on blocks proposed by honest leaders irrespective of inconsistencies introduced by an earlier Byzantine leader. This allows our protocol to always commit with a small constant latency in the good-case when the leader is honest.

Rotating sender chained reliable broadcast. We formalize the rotating-leader chained BFT SMR protocols in the form of rotating sender chained reliable broadcast as follows. Our lower bound is presented in the form of rotating sender chained reliable broadcast.

▶ **Definition 5** (Rotating Sender Chained Reliable Broadcast). *In a* rotating sender chained reliable broadcast*, there is a sequence of designated senders $p_1, \ldots, p_k$ for $k \geq 1$ sending messages such that when a sender p_i broadcasts B_x at some height x in the log, then the following conditions hold:*

1. *(Correctness) If some honest party q delivers message B_x at height x in the log, then eventually every honest party delivers B_x at height x in the log.*
2. *(Validity) If sender p_i is honest, then every honest party eventually delivers the input B_x that p_i broadcasts at height x.*
3. *(Sequentiality) If an honest sender p_j with $j < i$ sends message $B_{x'}$, then the input B_x sent by an honest sender p_i must extend $B_{x'}$.*
4. *(Extension) When an honest party q delivers a message B_x at height x in the log, it delivers all the ancestors of message B_x.*

Next, we formally define *optimistically responsive rotating leader chained reliable broadcast* as follows.

▶ **Definition 6** (Optimistically Responsive Rotating Sender Chained Reliable Broadcast). *We say that a rotating sender chained reliable broadcast is optimistically responsive if for any two consecutive honest senders p_i and p_{i+1}, p_{i+1} starts the broadcast in the sequence within $O(\delta)$ time after receiving p_i's input.*

3 A Lower Bound on the Good-Case Latency of Optimistically Responsive Rotating Sender Chained Reliable Broadcast

In this section, we provide a lower bound on the good-case latency of rotating leader protocols where the next leaders propose responsively without waiting $\Omega(\Delta)$ time after receiving proposals from previous leaders. In particular, the lower bound captures the relationship between latency of the view-change and the good-case latency of a commit. Essentially, it says that if a consensus protocol tolerating $n/3 \leq f < n/2$ Byzantine faults allows a new leader to propose responsively in α time, the commit latency for a single slot cannot be less than $2\Delta - \alpha$. Thus, the sum of latencies has to be at least 2Δ.

Unsynchronized starts. Our lower bound assumes an unsynchronized start model [3, 25] where honest parties may start the protocol execution at different times decided by an adversary such that the following conditions hold: (i) each honest party starts the protocol at time $\leq \Delta$ and (ii) an honest party starts the protocol before receiving a message from any other party. Such a model captures state machine replication protocols where replicas move to the next view/slot at different times. Byzantine parties, on the other hand, are assumed to start the protocol execution at time 0. The parties start the protocol with a fixed state independent of when the protocol execution started; in particular, they do not have access to the execution start time.

Intuition. Any Byzantine fault tolerant consensus protocol tolerating f Byzantine faults cannot wait to receive messages from more than $n - f$ parties. For a synchronous consensus protocol tolerating $f < n/2$, an honest party can wait for messages from at most $f + 1$ parties, assuming $n = 2f + 1$ for simplicity sake. Consider a set P of f honest parties and a set Q of

f honest parties. Suppose the current sender s_1 is Byzantine and sends some value B_e which arrives at parties in P at some time 0. The Byzantine sender s_1 also sends conflicting values B'_e to honest parties in Q which arrives only at time $\Delta - \alpha$. Observe that messages from a single Byzantine party s_1 is sufficient for parties in P to form a quorum of $f + 1$ messages. In addition, messages between parties in P and Q can be delayed by up to Δ time. Thus, parties in P may not learn about conflicting value B'_e before $2\Delta - \alpha$ time and will commit to value B_e before $2\Delta - \alpha$ time where $2\Delta - \alpha$ is the protocol commit latency.

The next sender $s_2 \in Q$ receives value B'_e at time $\Delta - \alpha$ and proposes value B_{e+1} by time Δ with α being the view-change latency. Note that sender s_2 may not learn about value B_e until time Δ. For sender s_2, the first sender s_1 appears to be honest until time Δ. Thus, by sequentiality property (refer Definition 5), honest sender s_2 extends B'_e and proposes value B_{e+1} by time Δ. Observe that due to Δ delay between messages exchanged between parties in P and Q, parties in P does not receive either B'_e or B_{e+1} before time $2\Delta - \alpha$ and hence cannot prevent parties in P to commit. Similarly, parties in Q does not receive value B_e before Δ and does not prevent the next sender s_2 from extending B'_e and proposing B_{e+1}. Since, our protocol always commits the value proposed by an honest sender, parties in Q eventually commit B_{e+1} and by extension property commit B_e. An important observation here is that if the commit latency were at least $2\Delta - \alpha$, parties in P would learn about value B'_e by time $2\Delta - \alpha$ and would not commit. This implies the sum of responsive view-change latency and good-case latency has to be at least 2Δ.

▶ **Theorem 7** (Lower bound on the good-case latency of optimistically responsive rotating sender sequenced reliable broadcast)**.** *For any $\alpha < \Delta$, there exists an execution in an optimistically responsive rotating sender chained reliable broadcast protocol in an unsynchronized start model tolerating $n/3 \leq f < n/2$ faults where the following two conditions do not hold simultaneously even in executions where messages between honest parties arrive instantaneously and all parties start at time 0: (i) the good-case commit latency is less than $2\Delta - \alpha$, and (ii) honest senders broadcast responsively in at most α time after receiving an input from the previous honest sender.*

Proof. Suppose for the sake of contradiction, there exists such a protocol. We will show a sequence of worlds, and through an indistinguishability argument show a violation in the last execution. Consider the parties being partitioned into the following 3 sets: (i) P: a set of f parties, (ii) Q: a set of f parties which includes the second sender s_2, and (iii) R: a set of $\max(1, n - 2f)$ parties which includes the first sender s_1.

World 1 (W1). *Setup.* Parties in $P \cup R$ are honest while parties in Q are crashed. An honest sender $s_1 \in R$ sends value B_e at some height e to all parties. Parties in P and R start at time 0.
Message schedule. Messages exchanged between parties in P and R arrive instantaneously.
Execution and views of honest parties. This execution satisfies (i), so parties in P commit B_e before $2\Delta - \alpha$ time.

World 2 (W2). *Setup.* Parties in $Q \cup R$ are honest while parties in P have crashed. An honest sender $s_1 \in R$ sends value B'_e at height e to all parties. Parties in Q and R start at time 0.
Message schedule. Messages exchanged between parties in Q and R arrive instantaneously.
Execution and views of honest parties. The next sender $s_2 \in Q$ receives B'_e at time 0. By sequentiality, the next sender s_2 extends B'_e and sends value B_{e+1} by time α. Since, sender s_2 is honest, by validity, parties in $Q \cup R$ eventually commit B_{e+1} at height $e + 1$. By extension, parties in $Q \cup R$ commit B'_e at height e.

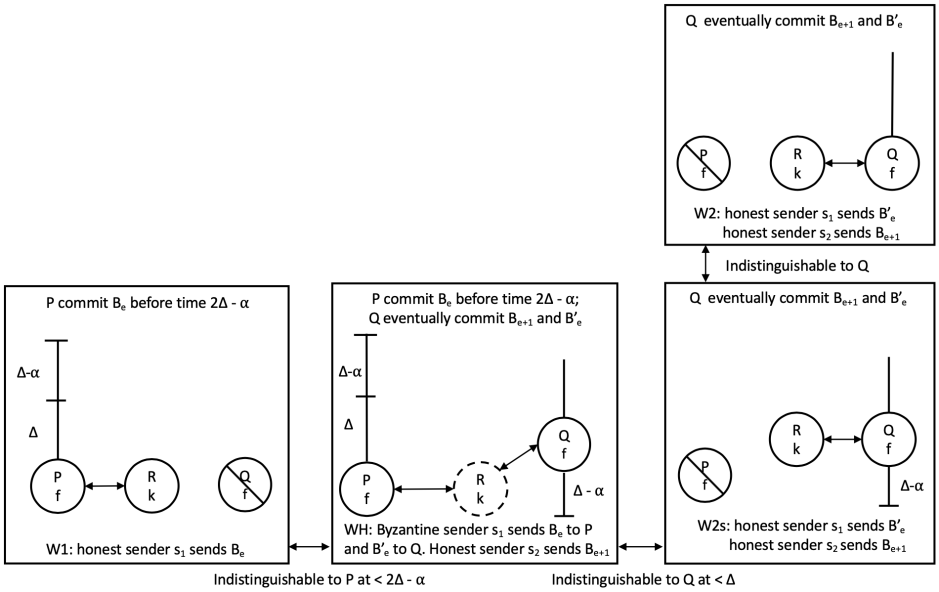


Figure 1 Latency of Optimistically responsive rotating sender chained reliable broadcast. Dotted circles represent Byzantine parties. Crossed circles represent crashed parties. Value of $k = n - 2f$.

World 2-shifted (W2s). *Setup.* Parties in $Q \cup R$ are honest while parties in P have crashed. An honest sender $s_1 \in R$ sends value B'_e to all parties. All parties start at time $\Delta - \alpha$.
Message schedule. Messages exchanged between parties in Q and R arrive instantaneously.

▷ Claim 8. Parties in $Q \cup R$ cannot distinguish between World 2 and World 2-shifted.

Proof. Observe that in an unsynchonized start model, parties start with a fixed state independent of when the protocol execution started and parties do not have access to the starting time. Moreover, all parties in $Q \cup R$ start at time $\Delta - \alpha$ in World 2-shifted. Thus, for parties in $Q \cup R$, this execution is indistinguishable to World 2. ◁

Execution and views of honest parties. By Claim 8, parties in $Q \cup R$ cannot distinguish between World 2 and World 2-shifted. Thus, the next sender s_2 sends value B_{e+1} by time $\Delta - \alpha + \alpha = \Delta$ by extending on B'_e. Since, this execution is identical to World 2, parties in $Q \cup R$ eventually commit B_{e+1} at height $e + 1$.

World Hybrid (WH). *Setup.* Parties in R are Byzantine. All other parties are honest. The Byzantine sender $s_1 \in R$ sends value B_e to parties in P and value B'_e to parties in Q. Parties in P start at time 0 while parties in Q start at time $\Delta - \alpha$.
Message schedule. Parties in P receive sender's value at time 0 while parties in Q receive sender's value at time $\Delta - \alpha$.

Parties in R perform a split-brain attack where they behave like World 1 towards parties in P, and behave like World 2-shifted towards parties in Q. We denote each brain of R as $R1$ and $R2$ such that $R1$ only communicate with P and $R2$ only communicate with Q.

Messages between parties in P and $R1$ arrive instantaneously (like in World 1). Parties in $R2$ sends messages to parties in Q only after time $\Delta - \alpha$ and messages between Q and $R2$ arrive instantaneously (like in World 2-shifted). Messages exchanged between parties in P and Q is delayed by Δ.

▷ Claim 9. The parties in P cannot distinguish between World 1 and World Hybrid until $2\Delta - \alpha$ time.

Proof. In World 1, parties in P start at time 0 and messages exchanged with parties in R arrive instantaneously. Since, parties in Q are crashed, parties in P do not receive any messages from parties in Q. Similarly, in World Hybrid, parties in P start at time 0 and messages exchanged with parties in $R1$ arrive instantaneously where parties in $R1$ behave identical to World 1 for parties in P. Moreover, since parties in Q start at time $\Delta - \alpha$ and messages between parties in P and Q are delayed by Δ, parties in P do not receive any messages from parties in Q until $2\Delta - \alpha$ time. Thus, parties in P cannot distinguish between World 1 and World Hybrid until $2\Delta - \alpha$ time. ◁

▷ Claim 10. The parties in Q cannot distinguish between World 2-shifted and World Hybrid until time Δ.

Proof. In World 2-shifted, parties in Q start at time $\Delta - \alpha$ and messages exchanged with parties in R arrive instantaneously. Since, parties in P are crashed in World 2-shifted, parties in Q receive no messages from parties in P. Similarly, in World Hybrid, parties in Q start at time $\Delta - \alpha$. Parties in $R2$ send messages only at time $\Delta - \alpha$ and messages between $R2$ and Q arrive instantaneously. Observe that parties in P start at time 0 and messages between P and R are delayed by Δ. Thus, parties in Q receive no messages from parties in P until time Δ. Thus, parties in Q cannot distinguish between World 2-shifted and World Hybrid until time Δ. ◁

Execution and views of honest parties. By Claim 9, parties in P cannot distinguish between World 1 and World Hybrid until $2\Delta - \alpha$ time. Thus, parties in P commit B_e by time $2\Delta - \alpha$ at height e.

By Claim 10, parties in Q cannot distinguish between World 2-shifted and World Hybrid until Δ time. Thus, the next sender $s_2 \in Q$ sends input B_{e+1} that extends B'_e by time Δ. By validity, parties in Q eventually commit B'_{e+1} at height $e + 1$. By extension, parties in Q also commit B'_e at height e. This violates correctness property between parties in P and Q at height e. A contradiction. ◀

4 Optimal Rotating-Leader BFT SMR with 2Δ-synchronous Latency

In this section, we present a rotating-leader chained BFT SMR protocol with optimal 2Δ-synchronous latency that allows *responsive* leader rotation while tolerating $f < n/2$ Byzantine faults. Prior synchronous protocols such as Sync HotStuff follow stable-leader paradigm and can make progress at network speed in the steady-state i.e., it can commit k proposals in $2\Delta + O(k\delta)$ where 2Δ is the commit latency for a single proposal. However, when the protocol changes leaders, they require a synchronous wait of $\Omega(\Delta)$ time during view-change process. Protocols such as OptSync [25] can perform responsive leader rotation. However, they require optimistic conditions where $> 3n/4$ replicas behave honestly. In the absence of optimistic conditions, they incur a synchronous delay of $\Omega(\Delta)$ time. In contrast, our protocol support responsive leader rotation in the absence of optimistic conditions while still tolerating $f < n/2$ Byzantine faults.

In addition, the commit step in our protocol is non-blocking i.e., we do not require replicas to commit before moving into higher epoch. Thus, our protocol can make progress at network speed while supporting responsive leader rotation and can commit k proposals in $2\Delta + O(k\delta)$ time with different leaders when there is a sequence of honest leaders. In this regard, our protocol can be viewed as rotating-leader version of Sync HotStuff.

Epochs. Our protocol progresses through a series of numbered *epochs* with each epoch e coordinated by a distinct leader L_e. Epochs are numbered by non-negative integers starting with 1. The leaders for each epoch are rotated irrespective of the progress made in each epoch. For simplicity, we use round-robin leader election and the leader of epoch e, represented as L_e, is determined by e mod n. The leaders can also be selected at random by using public known function such as random beacons [12] which allows leader election in $O(\delta)$ time. When the leader of an epoch is honest, the protocol progresses through epoch responsively, i.e., in $O(\delta)$ time; otherwise each epoch lasts for 7Δ time.

Blocks and block format. We represent a proposal in an epoch in the form of a *block*. Each block references its predecessor with the exception of the genesis block which has no predecessor. A block B_k is said to be *valid* if (i) its predecessor block is valid, or if $k = 1$, predecessor is $\perp$, and (ii) the payload in the block meets the application-level validity conditions. Note that a leader L_e proposes a single block in an epoch.

Certified blocks, and locked blocks. A block certificate on a block B_k consists of $t + 1$ distinct signatures in an epoch e and is represented by $\mathcal{C}_e(B_h)$. We define a simple ranking rule as leaders propose a single block in each epoch. Block certificates are ranked by epochs, i.e., blocks certified in a higher epoch has 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. Replicas will lock on highest ranked certified blocks and do not vote for blocks that do not extend highest ranked block certificates to ensure safety of a commit.

Block extension and equivocation. A block B_k *extends* a block B_l ($k \geq l$) if B_l is an ancestor of B_k. Note that a block B_k extends itself. Two blocks B_k and $B'_{k'}$ proposed in the same epoch *equivocate* one another if they are not equal.

4.1 Protocol Details

Protocol 1 Rotating-leader BFT SMR.

For each epoch $e = 1, 2, \cdots$:, replica r performs following operations:

1. **Epoch Advancement.** When epoch-timer$_e$ reaches 0, broadcast $\langle \mathsf{clock}, e + 1 \rangle_r$. Replica r advances to epoch $e + 1$ using following rules:
   - On receiving $f + 1$ votes for epoch e block.
   - On receiving $f + 1$ epoch e clock messages.

   Upon entering epoch $e + 1$, broadcast an epoch e certificate and send highest ranked block certificate to L_{e+1}, set epoch-timer$_{e+1}$ to 7Δ and start counting down. Abort epoch-timer$_e$.
2. **Propose.** If leader L_e of epoch e has $\mathcal{C}_{e-1}(B_l)$ propose immediately; otherwise wait for 2Δ time. Broadcast $\langle \mathsf{propose}, B_k, e, \mathcal{C}_{e'}(B_l) \rangle_L$ where B_k extends highest certified block B_l known to L_e.
3. **Vote.** Upon receiving the first proposal $\langle \mathsf{propose}, B_k, e, \mathcal{C}_{e'}(B_l) \rangle_L$, if B_k extends the highest ranked certificate known to replica r, broadcast a vote in the form of $\langle \mathsf{vote}, B_k, e \rangle_r$.
4. **(Non-blocking) Commit.** If epoch-timer$_e$ $> 2\Delta$ and replica r receives $\mathcal{C}_e(B_k)$, set commit-timer$_e$ to 2Δ and start counting down. When commit-timer$_e$ reaches 0, if no epoch-e equivocation has been detected, commit B_k and all its ancestors.
5. **Equivocation.** Forward the equivocating hashes signed by L_e and abort commit-timer$_e$. While still in epoch e, broadcast $\langle \mathsf{clock}, e + 1 \rangle_r$.

Our protocol (refer Protocol 1) is simple and includes following five steps for an epoch e.

Epoch advancement. Each replica r keeps track of epoch duration $\mathsf{epoch\text{-}timer}_e$ for epoch e. When its $\mathsf{epoch\text{-}timer}_{e-1}$ expires, replica r broadcasts an epoch e clock message, i.e., $\langle \mathsf{clock}, e \rangle_r$ to all replicas. Replica r enters epoch e when it receives either $f+1$ distinct $\langle \mathsf{clock}, e \rangle$ messages (i.e., an epoch e clock certificate) or when it receives an epoch e block certificate $\mathcal{C}_e(B_l)$ for some block B_l. Upon entering epoch e, replica r broadcasts an epoch e certificate (either clock certificate or block certificate) to perform epoch synchronization among all honest replicas. Replica r also sends its highest ranked certificate $\mathcal{C}_e(B_l)$ to the leader L_e if it sent a clock certificate while entering epoch e. In addition, it aborts all timers below epoch e and sets $\mathsf{epoch\text{-}timer}_e$ to 7Δ and starts counting down.

Propose. Upon entering epoch e, if Leader L_e has an epoch $e-1$ block certificate, it proposes immediately; otherwise, it waits for 2Δ time to ensure it can receive the highest ranked certificate from all honest replicas. The leader L_e proposes a block $B_k := (b_k, H(B_{k-1}))$ by broadcasting $\langle \mathsf{propose}, B_k, v, \mathcal{C}_{e'}(B_l) \rangle_{L_e}$ where $\mathcal{C}_{e'}(B_l)$ is the highest ranked certificate known to L_e.

Vote. When a replica r receives the first proposal for B_k either from L_e or through some other replica, if r hasn't received a proposal for an equivocating block, i.e., it has not detected a leader equivocation in epoch e, it forwards the proposal to all replicas. If block B_k extends the highest ranked certificate known to replica r, it broadcasts a vote for B_k in the form of $\langle \mathsf{vote}, B_k, e \rangle_r$.; otherwise, replica r does not vote for the proposed block. Voting for blocks that extends the highest ranked certificate ensures safety of committed blocks.

Commit. When replica r receives $f+1$ distinct vote messages for an epoch e block B_k (denoted by $\mathcal{C}_e(B_k)$) and when $\mathsf{epoch\text{-}timer}_e$ is large enough (2Δ), it sets its $\mathsf{commit\text{-}timer}_e$ to 2Δ and starts counting down. Note that replica r moves to epoch $e+1$ as soon as it receives $\mathcal{C}_e(B_k)$ while its $\mathsf{commit\text{-}timer}_e$ for block B_k is still running. When $\mathsf{commit\text{-}timer}_e$ reaches 0, if no equivocation for epoch-e has been detected, replica r commits B_k and all its ancestors. Waiting for 2Δ wait before commit ensures no honest replica has voted for an equivocating block in epoch e. In addition, honest replicas start their $\mathsf{commit\text{-}timer}_e$ only when their $\mathsf{epoch\text{-}timer}_e \geq 2\Delta$. This ensures no honest replica entered an higher epoch without receiving $\mathcal{C}_e(B_k)$.

Equivocation. At any time in epoch e, if a replica r detects an equivocation, it broadcasts equivocating hashes signed by leader L_e along with an epoch e clock message. Replica r also stops performing epoch e operations.

How does our protocol support responsive leader rotation? In general, consensus protocols require that all honest replicas receive and lock on a unique certificate for a block to be committed in an epoch before moving to higher epoch and not vote for blocks that do not extend this unique certificate to ensure safety of a commit. Prior synchronous protocols such as Sync HotStuff [2] achieve this property by adding a synchronous wait of at least 1Δ while moving to higher epoch. In Sync HotStuff [2], a replica starts its $\mathsf{commit\text{-}timer}$ of 2Δ as soon as it votes for the proposed block and commits when it does not detect any equivocation in the epoch before its $\mathsf{commit\text{-}timer}$ expires. While the replica that committed may not have detected an equivocation before its commit, other honest replicas might have detected

an equivocation. Waiting for Δ time before moving to higher epoch ensures all honest replicas receive at least $f+1$ votes for the committed block. However, a synchronous wait of full Δ during epoch-change makes protocols non-responsive. To achieve responsive epoch-change, our protocol instead waits for a certificate for the proposed block before starting the commit-timer and forwards the received certificate. Thus, replicas can responsively receive the certificate. In addition, a single block is proposed in an epoch. Thus, if the next leader receives the certificate for the current proposed block, it can propose immediately as it is already the highest ranked block certificate and all honest replicas will vote for the block extending this certificate. Initiating the commit-timer only upon receiving a certificate for the proposed block has also been explored in prior protocols such as Flexible BFT [19] which works in hybrid model (with both synchronous and partial synchronous assumptions) and follows stable-leader approach. In contrast, our protocol follows rotating-leader approach under synchrony assumption.

How does 2Δ wait ensure safety? Consider an honest replica r that commits a block B_k in epoch e at time t. Replica r must have received $\mathcal{C}_e(B_k)$ at time $t-2\Delta$ and did not detect any block e equivocation by time t. This implies no honest replica either received or voted for an equivocating block in epoch e by time $t-\Delta$; otherwise, replica r would have detected an epoch e equivocation by time t. In addition, since all honest replicas receive proposal for B_k by time $t-\Delta$, no honest replica will vote for an equivocating block in epoch e. This ensure the certificate for block B_k is unique. In addition, since replica r committed in epoch e, its epoch-timer$_e$ must have been epoch-timer$_e > 2\Delta$ at time $t-2\Delta$. Since, honest replicas are synchronized withing Δ time, honest replicas that are still in epoch e must have epoch-timer$_e > \Delta$ by time $t-2\Delta$. Replica r broadcasts $\mathcal{C}_e(B_k)$ at time $t-2\Delta$ when it starts its commit-timer at $t-2\Delta$. Thus, these replicas will receive $\mathcal{C}_e(B_k)$ before entering epoch $e+1$. This ensures all honest replicas receive $\mathcal{C}_e(B_k)$ before entering epoch $e+1$ and hence, no honest replica will vote for blocks that do not extend $\mathcal{C}_e(B_k)$. This ensures safety of committed block B_k.

Note that it is not required for all honest replicas to commit block B_k in the same epoch e. A Byzantine leader may send equivocating blocks to other honest replicas after time $t-\Delta$ and replica r may not receive such equivocation before its commit-timer expires. However, if an honest replica r commits a block B_k, our protocol ensures that all honest replicas receive a unique certificate for B_k before entering epoch $e+1$ and no honest replica vote for blocks that do not extend B_k in higher epoch. Eventually, due to leader rotation, there will an honest leader and this honest leader will propose block B_h extending B_k. All honest replicas will commit block B_h proposed by an honest leader and all honest replicas will commit B_k via ancestor rule (since B_h extends B_k).

▶ Remark. Our protocol can be extended with an additional commit rule that commits a block B_ℓ proposed in epoch $e-f$ at the end of epoch e if the highest ranked chain in epoch e extends B_ℓ, assuming there is a set of unique $f+1$ leaders in the last $f+1$ epochs. Such a commit rule can be used to commit faster when there is a set of $f+1$ consecutive honest leaders and progressing through $f+1$ epochs is faster than waiting for 2Δ time and obtain better commit latency.

The rationale behind the commit rule is the following. Out of last $f+1$ honest leaders, there will at least one honest leader between epochs $e-f$ and e. Consider an epoch e' (with $e-f<e'<e$ and its honest leader $L_{e'}$. The leader $L_{e'}$ will extend on the highest ranked chain certificate and propose some block B_k in a timely manner. Assume B_k extends B_ℓ. Thus, all other honest replicas will vote for the proposed block B_k and all honest replicas will receive and lock on $\mathcal{C}_{e'}(B_k)$ before entering epoch $e'+1$. Hereafter, no honest replica will vote for a block that does not extend B_k, and all certified blocks after epoch e' must extend

B_k, i.e., a conflicting chain of rank higher than $\mathcal{C}_{e'}(B_k)$ cannot form. Thus, the highest ranked chain in epoch e must extend B_k and since B_k extends B_ℓ, B_ℓ is safe to commit. A recent work, Apollo [4] uses such a commit rule to obtain better latency when there are a sequence of consecutive honest leaders. In their work, the protocol proposes without waiting for a certificate and can obtain $O(f\delta)$ latency with a sequence of honest leaders. In contrast, our protocol requires waiting for a certificate to obtain a good-case latency of 2Δ.

Due to space constraints, we present security analysis in Appendix A.

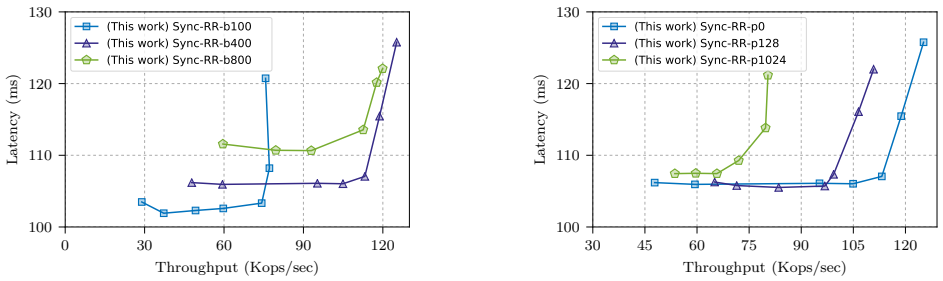
5 Evaluation

In this section, we compare the performance of our protocol with state-of-art synchronous protocol Sync HotStuff in both stable leader mode and rotating-leader mode.

5.1 Implementation Details and Methodology

Our implementation is an adaption of the open-source implementation of Sync HotStuff [11]. We modify the core consensus logic to our protocol and extend it to support leader rotation. We also extend Sync HotStuff protocol to support leader rotation after each block proposal. In this modified version, the leader is rotated every 5Δ time (2Δ wait for the next leader before proposing in the new epoch, 2Δ time to commit a block and 1Δ wait during epoch-change.)

Each block consists of a batch of client commands. Each command contains a unique command identifier and an associated payload. The number of commands in a block determines its batch size. The throughput and latency results were measured from the perspective of external clients that run on separate machines from that of the replicas. The clients broadcast a configurable number of commands to every replica at certain configurable time interval. In all of our experiments, we ensure that the performance of replicas are not limited by lack of client commands.



(a) Varying batch sizes. **(b)** Varying payload.

Figure 2 Throughput vs. latency at varying batch sizes and payload at $\Delta = 50$ms and $f = 1$.

Experimental Setup. All our replicas and clients were installed on Amazon EC2 `c5.2xlarge` instances. Each instance has 8 vCPUs supported by Intel Xeon Platinum 8000 processors with maximum network bandwidth of upto 10Gbps. The network latency between two machines is measured to be less than 1ms. We used `secp256k1` for digital signatures in votes and a quorum certificate consists of an array of $f + 1$ signatures.

Baselines. We make comparisons with the state-of-the-art synchronous protocol (Sync HotStuff) in two modes: (i) Sync HotStuff with a stable leader which is the default protocol, and (ii) Sync HotStuff with rotating-leaders with each leader making one block proposal per epoch.

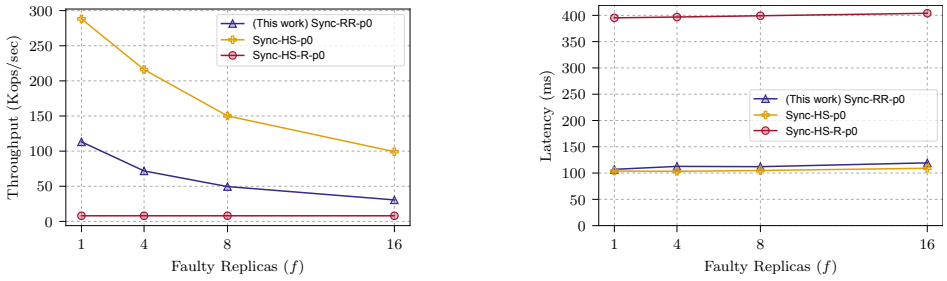
In general, stable leader protocols have better performance in terms of throughput. For fair comparision with our rotating-leader protocol, we also compare against a rotating-leader version of Sync HotStuff protocol where a new leader proposes a block every 5Δ time.

5.2 Basic Performance

We first evaluate the basic performance of our protocol when tolerating $f = 1$ fault at $\Delta = 50$ms. We measure the observed throughput (i.e., number of committed commands per second) and the end-to-end latency for clients. In our first experiment (Figure 2a), each command has a zero-byte payload and we vary batch size at different values: 100, 400, and 800 as represented by the three lines in the graph.

Each point in the graph represents the measured throughput and latency for a run with a given load sent by clients. The load is increased by sending more number of commands in a given time interval. As seen in the graph, the throughput increases with increasing load without increasing latency upto a certain point before reaching saturation. After saturation, the latency increases while the throughput either remains consistent or slightly degrades. We observe that the throughput is maximum at around 113 Kops/sec when the batch size is 400 with a latency of around 107ms. We set the batch size to be 400 for our following experiments.

In our second experiment (Figure 2b), we vary the command request/response payload at different values in bytes 0/0, 128/128 and 1024/1024 with a fixed batch size of 400. We observe that as the payload size increases, the throughput, measured in number of commands, decreases. We also observe a marginal drop in latency with increasing payload.



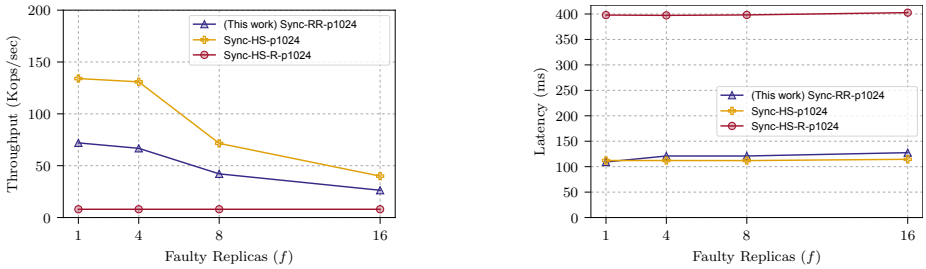
(a) f vs. throughput. **(b)** f vs. latency.

Figure 3 Performance as function of faults at $\Delta = 50$ms, 400 batch size, and 0/0 payload.

5.3 Scalability and Comparison with Prior Work

Next, we evaluate our protocol scales as the number of replicas increase. We compare our protocol against standard Sync HotStuff (Sync-HS) and rotating-leader version of Sync HotStuff (Sync-HS-R). First, we evaluate the protocol performance with zero-payload commands to understand the raw overhead incurred by the underlying consensus mechanism at different values of f (Figure 3). Then, we study how the protocols perform at a higher payload of 1024/2024 (Figure 4). We use a batch size of 400 and Δ of 50ms for both these experiments. For Sync-HS-R, we use a batch size of 2000. Each data point in the graphs represent the throughput and latency at the saturation point without overloading the replicas.

Comparison with Sync HotStuff. Figures 3 and 4 compares the throughput and latency of Sync Hotstuff with our protocol at two different payloads: 0/0 and 1024/1024. We observe the latency of the our protocol is similar to Sync HotStuff as both protocols have



(a) f vs throughput. **(b)** f vs latency.

Figure 4 Performance as function of faults at $\Delta = 50$ms, optimal batch size, and 1024/1024 payload.

2Δ commit latency. However, Sync HotStuff has much better throughput compared to our protocol. This is due to following reasons: (i) the performance of Sync HotStuff depends on the $f + 1$ fastest replicas in the system, (ii) being a stable-leader protocol, the leader can schedule block proposals as soon as sufficient commands to form a block are available. In contrast, the performance of our protocol is bottlenecked due to following reasons: (i) the performance depends on the slowest replica in the system due to round-robin leader rotation (ii) a leader can only schedule a block proposal after it has been elected as a leader, and (iii) since commands arrive in a different order at different replicas, an additional processing is required to filter out proposed commands (the additional processing incur around 150μs). Although, the stable-leader Sync HotStuff provides better throughput, it is worth to note the stable-leader approach does not provide fairness and censorship resistance. Next, we compare our protocol against Sync HotStuff with leader rotation after each block proposal which provides better fairness and censorship resistance.

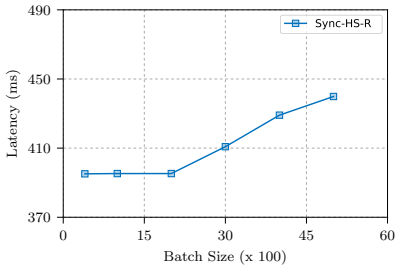


Figure 5 Latency of Sync HotStuff with leader rotation at varying batch sizes at $\Delta = 50$ms and $f = 1$.

Comparison with Sync HotStuff with leader rotation. In Sync HotStuff with leader rotation, a leader proposes a block every 5Δ time i.e., every 250ms at $\Delta = 50$ms. Thus, the throughput of the protocol is essentially 4 times the proposal batch size. We first perform an experiment (Figure 5) to find a batch size for which the latency does not adversely worsen. We observe that at the batch size of 2000, the latency is similar to the latency at batch size of 400. At higher batch sizes, the latency of the protocol worsens as can be seen in the figure. For other experiments (Figures 3 and Figure 4), we set the batch size to be 2000; thus the throughput of the protocol was always 8000. The latency of the protocol is also very high at around 400ms. This is because the leader proposes a block every 5Δ while clients sent commands much earlier. Since, our protocol supports responsive leader rotation, the next

leader can propose as soon it receive a certificate for previous block. Thus, our protocol performs much better compared to Sync HotStuff with leader rotation in terms of latency and throughput.

6 Related Work

There has been a long line of work in designing efficient BFT SMR protocol [10, 8, 25, 21, 4, 3, 2, 5]. Most of these BFT SMR solutions [2, 3, 25] focus in increasing the performance of the system during steady state and hence follow a stable-leader paradigm. Our work focuses in improving the performance of system while rotating-leaders every epoch as rotating-leader protocol provide better fairness and censorship resistance compared to stable leader protocols. We review the most recent and closely related works below. Compared to all of these protocol, our work commits in 2Δ time when the leaders are honest and the new leaders can propose responsively in $O(\delta)$ time. In addition, we do not require optimistic conditions to change leaders responsively.

The protocols due to PiLi [10] and Streamlet [8] provide BFT SMR protocols that change leaders after every epoch. Their leader selection is randomized. Both of these protocol assume lock-step execution in epochs. In PiLi, each epoch lasts for $O(\delta)$ (resp. 5Δ) under optimistic (resp. synchronous) conditions. PiLi commits 5 blocks after 13 consecutive honest epochs. PiLi has a responsive (resp. synchronous) latency of at least 16δ-26δ (resp. 40Δ-65Δ). Streamlet commits a single blocks after 6 consecutive honest epochs with each epoch taking 2Δ time. However, under $f < n/2$ corruption, getting 6 or 13 consecutive honest epochs is extremely unlikely and these protocols may never progress. In contrast, our protocol requires a single honest epoch to make progress.

The protocols due to OptSync [25] and Hybrid-BFT [21] provide rotating-leader BFT SMR protocols. However, they change leaders responsively only during optimistic conditions. During normal conditions when $f < n/2$ Byzantine faults are present, these protocol wait for at least 7Δ in an epoch before moving to the next leader even when there is a sequence of honest leaders. With a sequence of honest leaders, our protocol can progress through epochs in $O(\delta)$ time despite tolerating $f < n/2$ Byzantine faults.

Apollo [4] provides a rotating-leader BFT SMR protocol. In their protocol, the leader is rotating after every proposal in δ time and the proposed blocks are committed after $f + 1$ epochs irrespective of whether the leader is honest or Byzantine. If all the leaders in the next $f + 1$ epochs are honest, their protocol commits with a latency of $(f + 1)\delta$. However, when $f = O(n)$, even in the *good-case* $O(f)$ out of the next $f + 1$ leaders may be Byzantine, thus yielding a latency of $O(f\Delta)$ even when messages between honest parties are instantaneous. Thus, in the good-case, they fail to satisfy the first condition (latency of a single-slot) of our lower bound.

RandPiper [5] also provides a rotating-leader BFT SMR protocol. In their protocol, they present a BFT SMR protocol that achieve quadratic communication without using threshold signatures. However, it incurs 11Δ in every epoch and cannot progress responsively.

A recent work Internet Computer Consensus [6] provides a rotating leader BFT SMR in partially synchronous model. Similar to our work, their protocol also progresses to higher epoch as soon as a certificate is formed in the current epoch while the committing a block in the hindsight only when sufficient parties acknowledge the block certificate. In our protocol, we commit only when no equivocation is detected for 2Δ time after receiving a block certificate. In both cases, the protocols check to see if the block certificate is unique and sufficient parties have received the block certificate.

References

1 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, and Ling Ren. Dfinity consensus, explored. *IACR Cryptol. ePrint Arch.*, 2018:1153, 2018.

2 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 654–667, 2020.

3 Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of byzantine broadcast: A complete categorization. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 331–341, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3465084.3467899.

4 Adithya Bhat, Akhil Bandarupalli, Saurabh Bagchi, Aniket Kate, and Michael K Reiter. Apollo–optimistically linear and responsive smr. Cryptology ePrint Archive, Report 2021/180, 2021. URL: https://eprint.iacr.org/2021/180.

5 Adithya Bhat, Nibesh Shrestha, Aniket Kate, and Kartik Nayak. Randpiper – reconfiguration-friendly random beacons with quadratic communication. Cryptology ePrint Archive, Report 2020/1590, 2020. , To appear in ACM SIGSAC CCS 2021. URL: https://eprint.iacr.org/2020/1590.

6 Jan Camenisch, Manu Drijvers, Timo Hanke, Yvonne-Anne Pignolet, Victor Shoup, and Dominic Williams. Internet computer consensus. Cryptology ePrint Archive, Report 2021/632, 2021. URL: https://eprint.iacr.org/2021/632.

7 Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

8 Benjamin Y Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 1–11, 2020.

9 T-H Hubert Chan, Rafael Pass, and Elaine Shi. Pala: A simple partially synchronous blockchain. *IACR Cryptology ePrint Archive*, 2018:981, 2018.

10 T-H Hubert Chan, Rafael Pass, and Elaine Shi. Pili: An extremely simple synchronous blockchain. *IACR Cryptology ePrint Archive*, 2018:980, 2018.

11 Determinant. Sync-HotStuff. URL: https://github.com/hot-stuff/librightstuff.

12 Drand. Drand - a distributed randomness beacon daemon. URL: https://github.com/drand/drand.

13 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

14 Michael J Fischer, Nancy A Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986.

15 Matthias Fitzi. *Generalized communication and security models in Byzantine agreement*. PhD thesis, ETH Zurich, 2002.

16 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68, 2017.

17 Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series, consensus system. *arXiv preprint*, 2018. arXiv:1805.04548.

18 Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. In *Annual International Cryptology Conference*, pages 445–462. Springer, 2006.

19 Dahlia Malkhi, Kartik Nayak, and Ling Ren. Flexible byzantine fault tolerance. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1041–1053, 2019.

20 Zarko Milosevic, Martin Biely, and André Schiper. Bounded delay in byzantine-tolerant state machine replication. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, pages 61–70. IEEE, 2013.

21 Atsuki Momose, Jason Paul Cruz, and Yuichi Kaji. Hybrid-bft: Optimistically responsive synchronous consensus with optimal latency or resilience. *IACR Cryptol. ePrint Arch.*, 2020:406, 2020.

22 Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 3–33. Springer, 2018.

23 Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

24 Elaine Shi. Streamlined blockchains: A simple and elegant approach (a tutorial and survey). In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 3–17. Springer, 2019.

25 Nibesh Shrestha, Ittai Abraham, Ling Ren, and Kartik Nayak. On the optimality of optimistic responsiveness. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 839–857, 2020.

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

A Safety and Liveness

We say a block B_k is committed directly in epoch e if it is committed as a result of its own $\mathsf{commit\text{-}timer}_e$ expiring. We say a block B_k is committed indirectly if it is a result of directly committing a block B_l $(l > k)$ that extends B_k.

▷ Claim 11. If an honest replica enters an epoch e at time t, then all honest replicas enter epoch e by time $t + \Delta$.

Proof. Suppose an honest replica r enters epoch e at time t and broadcasts an epoch $e-1$ certificate. Suppose for the sake of contradiction, an honest replica r' does not enter epoch e by time $t + \Delta$. Since replica r broadcasts epoch $e-1$ certificate at time t, the epoch $e-1$ certificate arrives all honest replicas by time $t + \Delta$. This implies replica r' receives epoch $e-1$ certificate and moves to epoch e by time $t + \Delta$. A contradiction. ◁

▶ **Corollary 12.** *The epoch timers of two honest replicas may differ by up to Δ time.*

▷ Claim 13. If an honest replica broadcasts $\langle \mathsf{clock}, e+1 \rangle$ in epoch e, no honest replica directly commits a block in epoch e.

Proof. Suppose an honest replica r sends $\langle \mathsf{clock}, e+1 \rangle_r$ in epoch e. Replica r sends $\langle \mathsf{clock}, e+1 \rangle_r$ on two cases (i) when its $\mathsf{epoch\text{-}timer}_e$ expires before receiving epoch e block certificate, and (ii) when it receives an epoch e equivocation while still in epoch e.

Suppose for the sake of contradiction, an honest replica, say replica r', commits a block B_k in epoch e at time t. Since replica r' committed block B_k in epoch e at time t, it must have received a $\mathcal{C}_e(B_k)$ such that its $\mathsf{epoch\text{-}timer}_e \geq 2\Delta$ at time $t - 2\Delta$ and did not detect an epoch e equivocation by time t. By Corollary 12, replica r's $\mathsf{epoch\text{-}timer}_e$ must be $\geq \Delta$ at time $t - 2\Delta$ in the worst case. In addition, no honest replica detected an epoch e equivocation by time $t - \Delta$ and sent a $\langle \mathsf{clock}, e+1 \rangle_r$ due to epoch e equivocation; otherwise replica r' would have detected epoch e equivocation by time t and would not commit. All honest replicas will receive an epoch e certificate for B_k by time $t - \Delta$. Thus, replica r would not send $\langle \mathsf{clock}, e+1 \rangle_r$ in epoch e. A contradiction. ◁

▶ **Corollary 14.** *If a clock certificate exists in epoch e, no honest replica directly commits a block in epoch e.*

▶ **Lemma 15.** *If an honest replica commits a block B_k in epoch e, then (i) an equivocating block certificate does not exist in epoch e (ii) every honest replica receives $\mathcal{C}_e(B_k)$ before entering epoch $e+1$.*

Proof. Suppose an honest replica r directly commits an epoch-e block B_k at time t. Replica r must have received a $\mathcal{C}_e(B_k)$ at time $t-2\Delta$ such that its $\mathsf{epoch\text{-}timer}_e \geq 2\Delta$. All honest replicas receive the proposal for $\mathcal{C}_e(B_k)$ by time $t-2\Delta$.

For part (i), observe that after time $t-\Delta$, no honest replica will vote for an equivocating block in epoch e. If an honest replica voted for an equivocating block $B'_{k'}$ before $t-\Delta$ in epoch e, replica r would have received the equivocating proposal for $B'_{k'}$ by time t and would not commit. This contradicts the hypothesis of r committing B_k directly in epoch e. Therefore, an equivocating block will not get any honest vote and will not be certified in epoch e.

By part(i) of the Lemma, there does not exists an equivocating block certificate and by Corollary 14, epoch-e certificate must be a block certificate. Thus, all honest replicas receive $\mathcal{C}_e(B_k)$ and enter epoch $e+1$. ◀

▶ **Lemma 16** (Unique Extensibility)**.** *If an honest replica directly commits B_e in epoch e, then any certified block that ranks higher than $\mathcal{C}_e(B_k)$ must extend B_e.*

Proof. The proof is by induction on epochs $e' > e$. For an epoch e', we prove that if $\mathcal{C}_{e'}(B_{k'})$ exists then it must extend B_k . A simple induction shows that all later block certificates must also extend B_k.

For the base case, where $e' = e+1$, the proof that $\mathcal{C}_{e'}(B_{k'})$ extends B_k follows from Lemma 15. The only way $\mathcal{C}_{e'}(B_{k'})$ forms is if some honest replica votes for $B_{k'}$ using Step 3 in Protocol 1. Honest replica votes in epoch e' only if it extends a highest certified block. By Lemma 15, there does not exist an equivocating block certificate in epoch e and all honest replicas receive $\mathcal{C}_e(B_k)$ before entering epoch $e+1$. Thus, no honest will vote for a block that does not extend B_k.

Given that the statement is true for all epoch below e', the proof that $\mathcal{C}_{e'}(B_{k'})$ extends B_k follows from the induction hypothesis because the only way such a block certificate forms is if some honest replica votes for it. Since honest replicas vote in epoch e' with a valid epoch $e'-1$ certificate and by induction hypothesis on certificates of epoch $e < e" < e'$, $\mathcal{C}_{e'}(B_{k'})$ must extend B_k. ◀

▶ **Theorem 17** (Safety)**.** *Honest replicas do not commit conflicting blocks for any epoch e.*

Proof. Suppose for the sake of contradiction two distinct blocks B_k and B'_k are committed in epoch e. Suppose B_k is committed as a result of $B_{k'}$ being directly committed in epoch e' and B'_k is committed as a result of $B'_{k''}$ being directly committed in epoch e''. Without loss of generality, assume $k' < k''$. Note that all directly committed blocks are certified. By Lemma 16, $B'_{k''}$ extends $B_{k'}$. Therefore, $B_k = B'_k$. ◀

▶ **Theorem 18** (Liveness)**.** *All honest replicas keep committing new blocks.*

Proof. Each epoch lasts for 7Δ time. If the leader is Byzantine and does not propose any blocks or proposes equivocating blocks, an epoch change will trigger and change the leader. Due to round robin leader election, there will be an honest leader.

Consider an honest epoch e and its leader L_e. Let t be the time when the first honest replica enters epoch e. By Claim 11, all honest replicas enter epoch e by time $t+\Delta$. If L_e has $\mathcal{C}_{e-1}(B_l)$, it proposes immediately, otherwise it waits for 2Δ to receive the highest ranked block certificate $\mathcal{C}_{e'}(B_l)$. In any case, L_e proposes by time $t+3\Delta$ which arrives all honest replicas by time $t+4\Delta$.

Since, leader L_e is honest, it proposes a block B_k that extends highest ranked certificate $\mathcal{C}_{e'}(B_l)$, all honest replicas will vote for it by time $t + 4\Delta$. Thus, all honest replicas will receive $\mathcal{C}_e(B_k)$ by time $t + 5\Delta$ and move to epoch $e + 1$. Observe that $\mathsf{epoch\text{-}timer}_e \geq 2\Delta$ for all honest replicas by time $t + 5\Delta$. Thus, all honest replicas start their $\mathsf{commit\text{-}timer}_e$. Since leader L_e is honest, an epoch e equivocation does not exist. Thus, all honest replicas will commit. ◀