Apollo – Optimistically Linear and Responsive SMR

Adithya Bhat[†] Akhil Bandarupalli[†] Saurabh Bagchi[†] Aniket Kate[†]
Michael K Reiter[‡]

[†]Purdue University
[‡]Duke University

{abhatk, abandaru, sbagchi, aniket}@purdue.edu, michael.reiter@duke.edu

Abstract

Existing Byzantine fault-tolerant (BFT) state-machine replication (SMR) protocols in the standard (bounded) synchrony and weak synchrony models rely on equivocation detection to ensure safety. To perform a commit (or output a transaction block), this detection inherently requires $O(n^2)$ communication overhead among n nodes and waiting for $O(\Delta)$ time, where Δ is the worst-case network delay. The quadratic communication overhead limits scalability. Moreover, as the *typical* network latency δ tends to be much smaller than Δ, 51% honest-majority (and hence synchronous or weakly synchronous) solutions become slow as compared to 67% honest-majority asynchronous protocols working at network speed.

The observation that SMR commits do not have to be treated separately motivates this work. We propose *UCC (Unique Chain Commit)* rule, a novel yet simple rule for hash chains where extending a block by including its hash is treated as a vote for the block and all its direct and indirect parents. When a block obtains $f+1$ such votes, where f is the maximum number of faulty nodes in the system, we commit the block and its parents. We use this UCC rule to design *Apollo*, an SMR protocol with rotating leaders which has a linear communication complexity. Apollo proposes and commits a block every δ time units with every block having a latency of $(f+1)\delta$. When compared to existing works which use equivocation detection, we improve the optimistic commit latency when $(f+1)\delta < 2\Delta$. We prove the security of our protocol in both standard and weak synchrony models. We also implement and compare Apollo with the state of the art protocol, and demonstrate Apollo having commit latencies independent of and less than the Δ parameter, and also show significant gains in response rates with increasing n. For instance, when $n = 3$, Apollo can commit blocks of size 2000 as fast 3 ms. For $n = 65$, Apollo produces 3x more committed transactions per second than the state of the art Sync HotStuff protocol.

1 Introduction

In fault-tolerant distributed computing, State Machine Replication (SMR) [27] is a fundamental building block that is receiving renewed attention due to its potential to support blockchains. At its core, an SMR protocol coordinates a set of nodes of a deterministic service so that, collectively, they implement the abstraction of a single, correct server, even when a subset of nodes turn malicious (or Byzantine). Most protocols perform this coordination using a *leader* that the others follow, though this leader can change by design or in response to faults.

All the nodes behave correctly during most of the lifetime of a Byzantine fault-tolerant SMR protocol. Therefore it is important for SMR protocols to be efficient (in terms of cryptographic

operations and communication complexity) in this optimistic case, i.e., when all the leader(s) are correct. With this motivation, we want an optimistically efficient SMR protocol that uses the worst case parameters (Δ), expensive communication (such as quadratic or cubic), and expensive cryptography (threshold signatures) only when needed, i.e., when things are not optimistic, or *bad*.

In the standard synchrony model, the parameter Δ bounds the worst-case delay for messages sent between any two correct nodes. This bound is a conservative estimate of message delivery times when the network is at its worst during adverse scenarios such as congestion or downtime of networking components. For instance, an NTP network surveys measure the delay between a node and its time-synchronization peer, which serve as an indirect measurement of network latencies. Minar [25] showed that the hosts experience a *mean* delay of 33 ms, a standard deviation of 115 ms, and 10^{-3}% of nodes encounter latencies as high as 10s. Therefore, we clearly observe a disparity between everyday/regular latency and the worst case latency, which influences the choice of Δ.

Recent synchronous SMR protocols [3, 28, 20, 12, 14] primarily use the lack of equivocation in $O(\Delta)$ time as a means to guarantee safety. Consider optimistic network conditions with the real network speed being δ, where $0 \approx \delta \ll \Delta$, and the leader(s) of all these protocols are correct. Still, every proposal experiences a delay of $O(\Delta) + O(\delta)$ to get committed, since we want to ensure that there are no equivocating proposals. We are sure that there is no equivocation only after hearing no complaints in 2Δ time. This discourages use of large Δ parameters as it affects everyday performance and encourages setting the protocol parameter Δ as close to δ as possible. However, setting Δ too small is risky and can result in the limit of f faulty nodes to be exceeded artificially, since the standard synchrony model treats any correct node that is unreachable within Δ time as faulty.

Guo et al. [20] proposed an alternate model called the *weak synchrony model* which relaxes the standard synchrony assumption. In this model, the Δ bound needs to hold true for only $n-f$ nodes in the system in any round of the protocol. The remaining f nodes consist of Byzantine nodes and nodes that experience network partitioning for an arbitrary (but finite) duration for whom the Δ bound can be violated. The weak synchrony model is a more practical model when compared to standard synchrony, since it allows correct nodes to go offline for a brief period, while not being classified as faulty. Despite being a stronger model, existing protocols [20, 14, 3, 28] in this model also have commit rules, and thus commit latencies, that are dependent on the Δ parameter.

Partially synchronous [9, 10, 19, 31, 13] and asynchronous protocols [17, 24] on the other hand, sacrifice fault tolerance by tolerating 33% Byzantine faults, but in return enjoy commit latencies of $O(\delta)$, where δ is the actual network speed, but more importantly, independent of Δ, if the network had one. These models make minimal to no assumptions on the network, and therefore manage to be *responsive*, i.e., the commit latency depends on receiving a number of messages, which depends on the actual network speed δ.

In this work, we first observe that the dependence on Δ mainly stems from the need to detect equivocation. We make a second observation that Proof-of-work systems [26, 30] (such as Bitcoin) have an informal rule that after observing 6 blocks extending a block B, the block B is deemed as final, i.e., the probability of the block B being rejected and replaced with another block by another correct node is negligible. We naturally ask ourselves, how many blocks do we need to observe before we are sure that a block B is final, in a permissioned network. It turns out that the answer is $f + 1$, where f is the maximum number of faults tolerated in the system, provided: (i) each of the $f + 1$ blocks are from unique nodes, and (ii) we use a hash chain, i.e., every block contains the hash of the previous block.

Based on these observations, we develop a responsive consensus rule called the *Unique Chain Commit (UCC)* rule, which theoretically commits one block every δ time, with a per-block commit latency of $(f + 1)\delta$. Our consensus rule and the associated protocol Apollo rely on Δ only to detect

Table 1: **Theoretical comparison of the best case of Apollo with best case of related works.** Here, best case means the leaders are always correct and the f faults are not the leaders.

Protocol	**Latency**	**# Signing**	**# Sig. Verifications**	**Comm. Compl.**	**Block Period**	**Network**	**No Lock Step**
Streamlet [12]	$12\Delta + 12\delta$	$O(n)$	$O(n^2)$	$O(n^2)$	$2\Delta + 2\delta$	Std. Sync	x
Dfinity [21, 2]	$6\Delta + 6\delta$	$O(n)$	$O(n^2)$	$O(n^2)$	$2\Delta + 2\delta$	Std. Sync	x
Opt Sync [28]	$2\Delta + 2\delta$	$O(n)$	$O(n^2)$	$O(n^2)$	2δ	Std. Sync	✓
Sync HS [3]	$2\Delta + 2\delta$	$O(n)$	$O(n^2)$	$O(n^2)$	2δ	Std. Sync	✓
PiLi [14]	$65\Delta + 2\delta$	$O(n)$	$O(n^2)$	$O(n^2)$	5Δ	Weak Sync	✓
Sync HS [3]	$2\Delta + 8\delta$	$O(n)$	$O(n^2)$	$O(n^2)$	2δ	Weak Sync	✓
Apollo	$(f+1)\delta$	$O(1)$	$O(n)$	$O(n)$	δ	Weak Sync	✓

Block Period is defined as the time between two successive block proposals.
 # Signing and # Sig. Verifications are the number of signature generation and verification operations performed by all the nodes in the system per proposal/block.
A protocol is said to have *No Lock step* if different nodes can be at different rounds at any point in time.

Table 2: **Theoretical comparison of the worst case of Apollo with the worst case of related works.** Here, worst case means f consecutive leaders (with view change for some protocols) crashed/equivocated.

Protocol	**Latency**	**Signature Complexity**	**Verification Complexity**	**Comm. Complexity**
Streamlet [12]	$O(f\Delta)$	$O(fn)$	$O(fn^2)$	$O(fn^2)$
Dfinty [21, 2]	$O^\star(f\Delta)$	$O(fn^2)$	$O(fn^3)$	$O(fn^3)$
Sync HS [3]	$O(p^\star\Delta) + O(f\Delta)$	$O(fn)$	$O(fn^2)$	$O(fn^2)$
Opt Sync [28]	$O(p^\star\Delta) + O(f\Delta)$	$O(fn)$	$O(fn^2)$	$O(fn^2)$
PiLi [14]	$O(f\Delta)$	$O(fn)$	$O(fn^2)$	$O(fn^2)$
Apollo	$O(f\Delta)$	$O(fn)$	$O(fn^2)$	$O(fn^2)$

$O^\star(g)$ denotes $O(g)$ with high probability.
$p^\star$ denotes the number blocks proposed before the leader crashes. In Sync HotStuff [3], a leader is blamed only if p blocks are not proposed in $2p + 4$ time. If p' blocks are proposed by time t, then the nodes wait for $p^\star = (2p' + 4 - t)$ time before blaming the leader.
Signature Complexity and *Verification Complexity* are the complexity of signature generation and verification operations.

a crashed leader. Asymptotically, for large enough n, $f\delta$ outgrows any fixed worst-case network delay Δ (since $f = \lfloor \frac{n-1}{2} \rfloor$ grows with n). However, if $\delta \ll \Delta$, or if the system size n is sufficiently small, then $f\delta < \Delta$ can hold true. Our UCC rule and Apollo protocol provides an alternative network-dependent commit latency of $(f+1)\delta$ over the existing fixed latency equivocation-based

protocols which have $O(\Delta)$ commit latencies. Our UCC rule and the Apollo protocol are secure in both standard and weak synchrony models. We also observe Apollo as being highly suitable for CPS settings. According to Table 1, in the best case, every node running Apollo signs and verifies only 1 digital signature per block and has a communication complexity of $O(n)$ per block, both of which contribute to energy efficiency.

1.1 Novelty and Key ideas

UCC (Unique Chain Commit) Rule. Consider a permissioned system with blocks proposed in every round by different nodes. At a high level, we use the idea that if a block B proposed in round r includes the hash of its parent block from the previous round $r-1$, then it implicitly votes for the chain consisting of parent blocks linked by their hashes. In SMR protocols so far, quorum certificates (i.e., a vector of signatures of more than 50% or 66.67% of the nodes in the system), were always built for every block of every round. We instead treat blocks proposed in subsequent rounds $\{r+1, \ldots, r+f+1\}$ from different leaders, as a certificate for the block proposed in round r as long as they are linked to each other through a hash chain. We term this rule as the Unique Chain Commit (UCC) rule.

Linearity. Recent synchronous SMR protocols [3, 28, 14, 12] require $O(n^2)$ communication in both the standard and weak synchrony models, even when all the nodes are correct. The quadratic communication is fundamentally necessary in these schemes as every step involves multicasts, so that the equivocation of proposals can be detected. We overcome this by using a round-based pipeline in the Apollo protocol, where in every round, a new leader (different from the last f leaders) proposes a block after receiving the block from the previous leader. Since we only need a block from the previous leader to propose a block, we achieve linear communication complexity in the optimistic case, which is also sufficient to handle equivocations by the leader.

Responsiveness. In the optimistic cases, i.e., when all the proposers are correct, our UCC rule allows Apollo to commit at network speed, i.e., at speeds independent of Δ. Apollo commits a block every δ time, with a latency of $(f+1)\delta$ per block. We define *block period* as the time between two successive block proposals in a protocol. Our Apollo protocol needs $f+1$ blocks to extend a chain to commit the chain, leading to a per-block latency of $(f+1)\delta$. Since, in every round, a new block gets $f+1$ extensions, we commit a block every δ time and thus have a block period of δ. This is another benefit of not depending on equivocation detection.

Relative timers. Existing works either use fixed epochs or rounds [12, 14, 20] or *view-relative* blame timers [3, 28], where the blamer timer counts the number of proposals received since the last view change. Our UCC rule allows Apollo to use *round-relative* blame timers which are blame timers relative to the last proposal, and hence the latest round. When a node receives a block for round r it can send it to the leader of round $r+1$ and wait for 2Δ.

Synchronization protocol. A block in Apollo requires f more blocks to be proposed after it, in order to commit it. If n (and therefore f) is large or if $\delta \approx \Delta$, then blocks suffer high latency. Some specific transactions could be time-sensitive, or some clients may require faster commits. We present a chain synchronization protocol that achieves this (Section 5). We show that this can be achieved by using any Byzantine Agreement protocol for the standard or weak synchrony models.

1.2 Summary of related works

Recently, several SMR/blockchain protocols [10, 14, 13, 31, 3, 12, 20] have emerged, in the standard synchrony, weak synchrony and partial synchrony models. For instance, Proof-of-Stake (PoS)

blockchain protocols require a rotating leader based consensus protocol, where the leader is chosen randomly, and whose probability of being a leader for an epoch/round is directly proportional to the amount of stake invested by the node. Therefore, permissioned consensus protocols are of interest in this area. In our literature review, we focus on works that are similar to our work and use standard synchrony or weak synchrony assumptions. We give a summary of related works and compare it with this work in Table 1 and Table 2.

Guo et al. [20] first proposed the weak synchrony[1]model. They showed how to achieve Byzantine Agreement in that model. Later, PiLi [12] proposed an improved SMR protocol in this model. The latest work, Sync HotStuff [3] also proposes a protocol in the weak synchrony model, but it calls the model the mobile sluggish fault model.

Sync HotStuff [3], Dfinity [21, 2], Streamlet [12] and PiLi [14], are SMR protocols with standard and weak synchrony network assumptions. Sync HotStuff [3] provides three protocols, one of which is an SMR protocol for weak synchrony. The original Dfinity protocol [21] is a white-paper that does not give all the details for the protocol. The details of the protocol are embedded in their implementation which is not yet open sourced. Abraham et al. [2] give a non-lock step version of Dfinity consensus that we analyze and use to compare with Apollo. Streamlet [12] includes blockchain protocols for the partial and standard synchrony models. We use their standard synchrony variant for comparison. PiLi [14] is a blockchain protocol designed for the weak synchrony model.

Sync HotStuff. Sync HotStuff [3] proposes three protocols: (a) an SMR protocol for standard synchrony, (b) an SMR protocol for mobile sluggish faults, and (c) an SMR protocol with optimistic responsiveness in the mobile sluggish fault model. They use the term *mobile sluggish fault model* instead of weak synchrony as defined by Guo et al. [20]. Their protocol uses a fixed leader and runs in views. In a view, the leader can propose as many blocks as it wishes as soon as it has a certificate for the latest block. This gives rise to the extra δ between two successive block proposals. Their protocol for weak synchrony requires a certificate on the successor of the block, and $f + 1$ `PRE-COMMIT` messages before starting the 2Δ timer, all of which gives rise to a theoretical latency of $2\Delta + 8\delta$ time after proposing to commit a block. Since all of these steps occur in parallel, two consecutive proposals are only delayed by 2δ.

In contrast, Apollo outputs two blocks for every block proposed by Sync HotStuff, with a net of 100% theoretical improvement in throughput. Comparing block latencies with Sync HotStuff, Apollo performs worse than Sync HotStuff when $f\delta > 2\Delta$ and so when $n > \frac{4\Delta}{\delta}$. For example, if $\delta = 1$ms and $\Delta = 1$s, then for $n > 4000$, we theoretically start having worse block latencies than Sync HotStuff. We find that Sync HotStuff is the state of the art synchronous SMR protocol for both standard synchrony as well as the weak synchrony model and therefore treat is as our baseline.

PiLi. PiLi [14] proposes a blockchain (SMR) protocol, specifically for the weak synchrony model. Rounds in PiLi are called *epochs*. Every epoch consists of one propose and one vote step. Each epoch r lasts for 5Δ as stated explicitly by Chan et al. [14], if the leader of epoch $r + 1$ cannot get a strongly notarized block (greater than $\frac{3}{4}n$ votes) but only a notarized block (greater than $\frac{n}{2}$ votes). This leads to a block period of 5Δ between two successive proposals. However, the commit rule that is employed is: after observing 13 consecutive notarized (certified with $> f$ votes) blocks, commit the prefix after removing the top 8 blocks. The commit rule also states that before voting it must observe that there is no conflicting notarization for a block with the same epoch number.

[1]Guo et al. [20] introduced the χ-weak synchrony model, where χ is the fraction of nodes that are online at any given time in the system. They show that the condition $\chi \geq 0.5$ is necessary and sufficient to achieve Byzantine Agreement. We refer to this model as the weak synchrony model. PBFT [11] also defines another weak synchrony model, but it is not used in any other literature.

This gives an additional 2Δ time before two successive proposals. Since we now require 13 blocks to be notarized before committing a block, we therefore must wait for 13 epochs, giving rise to the numbers in Table 1.

All of the above synchronous (both standard and weak synchrony) SMR protocols have one thing in common: detect equivocation and commit after ensuring that there is no equivocating blocks or proposals. Their influence can be clearly observed in the quadratic communication and quadratic signature verification complexity in Table 1.

On optimistic responsiveness. Sync HotStuff [3], OptSync [28] and PiLi [14] support a mode called *optimistic responsiveness*. In this mode, they assume $> \frac{3}{4}n$ nodes along with the leader(s) are correct and the network delivers messages for the correct nodes in $O(\delta)$, which allows the protocols to commit in $O(\delta)$. This is a different assumption that is made on the system and the leader. However, when compared to an optimistically responsive protocol, our protocol suffers a worse block latency, but we provide a trade-off between latency and throughput (block period).

1.3 Contributions

This work makes the following contributions:

1. We develop a novel consensus rule called *UCC (Unique Chain Commit)*, which avoids the need to detect equivocation. This rule is allows us to go beyond the minimum commit latency of $O(\Delta)$ in optimistic scenarios of correct leaders, concretely when $(f+1)\delta < 2\Delta$, where δ is the real or actual network speed and Δ is the worst case network delay.

2. We develop Apollo, an optimistically linear SMR protocol for standard and weak synchronous models using the UCC rule. It has the following features: (i) The Apollo protocol has $O(n)$ communication complexity when all the leaders are correct and $O(n^2)$ when a leader crashes or equivocates. (ii) Apollo is efficient in terms of cryptographic operations. It uses 1 signature generation and $n-1$ signature verification operations per round in total for all the nodes in the system. This is critical for resource-constrained devices where energy is limited. (iii) In the optimistic cases, Apollo commits one block every δ time and every block has a latency of $(f+1)\delta$. (iv) Apollo uses round-relative timers where blame timers are based on the last received block, instead of view-relative blame timers based on the number of proposals received since the last view change or fixed $O(\Delta)$ rounds.

3. We also present a synchronization protocol that allows nodes to commit to bypass the UCC rule and safely commit a consistent chain, i.e., commit the block with the largest height, possibly for time-sensitive transactions or periodically, without waiting for $f+1$ suffix blocks. We use a black-box Byzantine Agreement protocol in the standard or weak synchrony model and show how to achieve synchronization without breaking safety of Apollo.

4. We implement both Apollo, Sync HotStuff [3] and Sync HotStuff (Round Robin) in Rust [6], to contrast our protocol with existing works. Our implementation of Sync HotStuff in Rust has better numbers ($20-30$ Kops/s more) for Sync HotStuff than reported in the original paper. We show that Apollo is the best round robin protocol, and its performance is comparable and close, within 3.33% of Sync HotStuff for $n=3$ and a block size of 2000 transactions with optimistic latencies of 3 ms which until now was achievable only by asynchronous protocol. For $n=65$, and a block size of 400 transactions, thanks to its linear computation and communication complexity, Apollo has a 3x better response rate (i.e., committed transactions per second) than Sync HotStuff. Finally, we find that our Rust code-base for both Sync HotStuff and Apollo to be a contribution to the community.

5. With its linear communication and signature complexities as compared to the quadratic signatures and communication complexities of Sync HotStuff, Apollo is highly suitable for energy-constrained environments. Towards demonstrating this capability, we also analyze Apollo in a battery-restricted distributed CPS setting where nodes communicate over Bluetooth low-energy (BLE) channels. We observe that Apollo consumes 4.5× and 8× lesser energy than Sync HotStuff for SMR for the leader respectively in the $n = 3$ and $n = 5$ CPS nodes system. For increasing values of n, this gap will increase significantly.

Paper Organization. The rest of the paper is organized as follows: Section 2 describes the system model, notation and definitions. Section 3 introduces the Unique Chain Commit (UCC) rule. Section 4 describes Apollo with details and security proofs. Section 5 introduces and describes the head synchronization protocol. Section 6 evaluates Apollo our implementation with the state-of-the-art SMR protocols for various system parameters. Section 7 discusses Apollo in the context of Cyber-Physical Systems (CPS).

2 Preliminaries

2.1 System Model

In this work, our system $\mathcal{N} := \{p_1, \ldots, p_n\}$ consists of n nodes participating in the SMR protocol, of which f may suffer Byzantine faults; we assume $n > 2f$. We refer to a node as *correct* in an execution if it never fails. We denote signed messages from p_i by $\langle \cdot \rangle_{p_i}$. We denote $f + 1$ signatures on the same message m as a certificate $\mathcal{C}(m)$. Similar to [21, 3, 28, 14, 13, 31], we assume a threshold (BLS [8]) signature scheme to reduce the size of the certificates to $O(1)$.

Network. We consider two network models: *standard synchrony* and *weak synchrony*.

1. *Standard synchrony* assumes that there is a known value Δ such that if a correct node sends a message to another correct node, then the message is received by the latter within Δ time from when it was sent by the former. We assume that faults in this model occur adaptively, i.e., the set of nodes chosen by the adversary to fail can grow monotonically up to f nodes as the protocol progresses.

2. *Weak synchrony* networks [20, 14, 3] follow standard synchrony for message delivery for all $> n/2$ nodes. At each time t, the network designates a set $\mathcal{O}_t \subseteq \mathcal{N}$ of correct nodes as *online*. The weak synchrony model entails the following assumptions. First, at every time t, at least $(n + 1)/2$ nodes are online; i.e., $|\mathcal{O}_t| > n/2$ for all times t. Second, if a correct node $p_i \in \mathcal{O}_{t_0}$ sends a message to correct node p_j at time t_0, then p_j receives this message sometime in the interval $(t_0, t_1]$, where t_1 is the earliest time $\geq t_0 + \Delta$ at which $p_j \in \mathcal{O}_{t_1}$. If such a time t_1 does not exist, then p_j might never receive the message.

We use the term *multicast* to mean a sendall operation where a node sends a message to all its connected nodes.

Delays. We use two delays in this work: Δ refers to the synchrony bound, i.e., the *worst case network delay*, and δ refers to the optimistic (actual/real) network speed[2].

[2]In the real world, the parameter δ varies between pairs of nodes, instances of time, and size of the message. However, for the theoretical analysis, we assume that a single δ value is the optimistic delay time, violation of which implies that we are not in the optimistic scenario.

2.2 SMR — State Machine Replication

In an SMR protocol (Definition 2.1), we wish to execute a state machine distributed across different nodes. The state of the distributed state machine must be consistent, i.e., no two correct nodes must output different states at any point. We instantiate the state machine using a distributed shared log. The log consists of sequences of blocks with inputs to the state machine. As long as the correct nodes agree on the ordering of blocks in the log, they will have a consistent state. We define clients as entities (possibly internal) that want to use, i.e., they provide inputs to, and/or obtain outputs from, the state machine.

Definition 2.1 (SMR — State Machine Replication [3])**.** *Assume a system of n nodes $\mathcal{N} := \{p_1, \ldots, p_n\}$, f of which are Byzantine faults.*

1. ***Safety.*** *If two correct nodes p_i, $p_j \in \mathcal{N}$ commit to blocks B_k and $B_k^\star$, respectively, at the same log height k, then $B_k = B_k^\star$.*

2. ***Liveness.*** *Each client request is eventually committed by all correct nodes.*

2.3 Blocks and Hash-chains

The nodes agree on a chain, which we define as a sequence of blocks, where blocks consist of commands (or transactions) from the clients. The commands are inputs to the state machine. We define the height of a block as the position of the block in this sequence or the chain. We define the first block as the genesis block to have a height 0. We assume that all the nodes use the same genesis block before starting the protocol. We denote a block at height k as B_k, and therefore the genesis block is B_0.

A block B_k at height k must always include the hash of the block B_{k-1} at height $k-1$. We define this hash as the parent hash or the parent pointer. We define B_{k-1} as the parent of B_k. We also define B_k as the child of B_{k-1}. We define a block $B_{k'}$ at height $k' < k$ as the ancestor of B_k, or an indirect parent of B_k, if we can reach $B_{k'}$ by following the parent pointers of B_k. We also define B_k as the indirect child of $B_{k'}$.

By default, the genesis block is defined to be always *valid*. We inductively define the validity of the child of a valid block. The child B of a valid block B' is valid, if B contains the hash of B' and it satisfies other validity conditions imposed by the state machine and the underlying protocol.

Finally, we define a valid chain $\mathcal{C} := \{B_0, \ldots, B_\ell\}$ as a sequence of valid blocks starting with the genesis block B_0. We define the chain to be of size ℓ if the highest height of blocks in the chain is ℓ. Note that we exclude the genesis block when counting the size.

Since, every block also includes the hash of its parent, we obtain a tamper-resistance property, i.e., given a valid chain of size ℓ, $\{B_0, \ldots, B_{k-1}, B_k, B_{k+1}, \ldots, B_\ell\}$ and particular block at height k, it is **not possible** to obtain another valid chain of blocks of the form $\{B_0, \ldots, B_{k-1}, B_k^\star, B_{k+1}, \ldots, B_\ell\}$ since changing a block naturally changes its hash thereby rendering all its direct and indirect children invalid members of the chain. In other words, it is not possible to change any ancestor of a block and we call this property as tamper resistance in Lemma 2.1.

Lemma 2.1 (Tamper resistance of hash-chains)**.** *Let $\mathcal{C} := \{B_0, \ldots, B_\ell\}$ and $\mathcal{C}^\star := \{B_0^\star, \ldots, B_\ell^\star\}$ be two valid chains of size ℓ. If $B_\ell = B_\ell^\star$, then $B_{\ell-1} = B_{\ell-1}^\star$, …, $B_0 = B_0^\star$, which implies $\mathcal{C} = \mathcal{C}^\star$.*

Given a chain $\mathcal{C} := \{B_0, \ldots, B_\ell\}$ we define B_ℓ as the head of the chain. Let $\mathcal{C}$ and $\mathcal{C}^\star$ be two chains of size ℓ and ℓ' respectively with $\ell < \ell'$. We define truncation of chain $\mathcal{C}^\star$ to size ℓ as $\mathcal{C}^\star[:\ell]$. We say that $\mathcal{C}$ is a prefix of $\mathcal{C}^\star$ if $\mathcal{C} = \mathcal{C}^\star[:\ell]$, and we denote that $\mathcal{C}$ is a prefix by using the set notation as $\mathcal{C} \subseteq \mathcal{C}^\star$. We define a common prefix chain $\mathcal{C}'$ for $\mathcal{C}$ and $\mathcal{C}^\star$ as the chain of size $\ell'' \leq \ell$, such that $\mathcal{C}[:\ell''] = \mathcal{C}^\star[:\ell'']$, and we denote using the set notation $\mathcal{C}' = \mathcal{C} \cap \mathcal{C}^\star$. Note that, for any two valid chains $\mathcal{C}$ and $\mathcal{C}^\star$, $\mathcal{C}_0 \in \mathcal{C} \cap \mathcal{C}^\star$, where $\mathcal{C}_0 := \{B_0\}$ contains the genesis block.

3 Unique Chain Commit (UCC) Rule

3.1 Certificates Revisted

In this section, we revisit quorum certificates. A quorum is a set of nodes, and a quorum certificate for a message consists of signatures by its members. In asynchronous SMR protocols (e.g., [10, 9, 31, 11, 12, 13]) quorum certificates with quorums of size $f+1$ and $2f+1$ are used. For instance, in Casper [10], a chain is finalized if there are more than $\frac{2n}{3}$ stakeholders (a $2f+1$ quorum certificate) who vote for it or its children.

Synchronous SMR protocols [3, 1, 28, 14, 12] typically improve the fault tolerance from $n > 3f$ to $n > 2f$ by detecting equivocation since every round terminates in a finite time. However, we observe the following about equivocation: (1) Equivocation is a chain fork (multiple valid chains), when hash-chains are used. (2) Resolving equivocations translates into a fork-resolution problem. Therefore, given two chains $\mathcal{C}$ and $\mathcal{C}^\star$, we need some weighting mechanism that determines which chain must be accepted by the correct nodes.

The chain weight problem. With these observations, it is clear that we need to determine a method to resolve forks. For a permissioned system in the standard and weak synchrony models tolerating f Byzantine nodes, a weight of $f+1$ votes (or an $f+1$ quorum certificate) on a block (and thus the parent chain) is insufficient to remove equivocation detection. Consider a Byzantine leader p_L. It can propose two blocks B and $B^\star$. If two correct nodes vote for B and $B^\star$ respectively, without being aware of the existence of the other block, then with the votes from the f Byzantine nodes, both the blocks can obtain sufficient votes and weight. This violates safety and therefore this weight mechanism is insufficient. We show how to solve this in the next subsection.

3.2 Commit Rule

Moving away from fixed leaders. For standard synchrony and weak synchrony, consider a round-robin method of selecting leaders. This has several interesting properties: (1) We are guaranteed one correct leader every $f+1$ rounds. (2) We are assured of one irreplacable correct proposed block in rounds where the leaders are correct. By irreplacable, we mean no node can prove that this block is false, such as by providing $f+1$ complaints. (3) The weight of the chain can be determined by ensuring that no alternate chain can get committed, i.e., the chain is also irreplacable.

New chain weights. Instead of counting the number of votes on a block as the weight of the block, we instead informally define the chain weight as follows:

> The *weight* of a chain is defined as the number of children for the head of the chain.

For permissioned systems in standard and weak synchrony models, when the weight of a chain prefix exceeds f, it is safe to commit the chain.

This is because all the correct nodes propose only block per height. There can exist multiple chains, unbeknownst to a correct leader, but when a correct leader proposes a block, it fixes the block at that height. At this point, all we need to do is to ensure that a correct proposal is always extended by correct nodes. We can ensure this by stating that a valid block can be rejected if there are $f+1$ explicit votes against it. By explicit votes, we mean $f+1$ signed messages (also known as blame messages) against the proposer of the block, the round, or the block itself.

Any node in such a system can be sure that the maximum length of a forked chain cannot exceed f, which occurs when f continuous leaders are Byzantine. In other words, for a given chain $\mathcal{C}$, only the suffix $\mathcal{C}_s$ consisting of the last f blocks can be different according to different correct nodes.

Rethinking certificates. Based on our earlier discussion on quorum certificates, we can look at a $f+1$ weighted chain as the $f+1$ quorum certificate for the chain. In all the consensus literature so far, certificates consisted of signatures on a particular message/block, and use $O(1)$ such quorum certified blocks in the commit rules. However, we look at $f+1$ proposals extending a hash-chain as a certificate to the chain.

Commit rule. Due to the interesting tamper resistance property (Lemma 2.1) provided by the hash chain, we now formally introduce our commit rule in Definition 3.1.

Definition 3.1 (UCC Rule)**.** *On observing any valid chain* $\mathcal{C} := \{B_0, \ldots, B_\ell\}$ *of size* ℓ *(with* $\ell > f$*), commit the prefix chain* $\mathcal{C}[: \ell - f]$*.*

Safety. A correct leader finalizes the hash-chain when it proposes a block. We ensure that correct proposals are always committed by the other nodes. Naturally, since the system only has f faults in a set of $\{B_i, \ldots, B_{i+f+1}\}$, at least one leader is correct and therefore pins at least one of the blocks (we do not know which). No other Byzantine node can produce a longer alternate chain because of the tamper resistance property. Therefore B_i is pinned (directly or indirectly) and it is always safe to commit block B_i and all its ancestors in the chain. We have done without trying to detect equivocation by purely knowing that there are only f faults in the system.

Discussion. The UCC commit rule has several advantages:

1. It is pipeline friendly, i.e., every new block adds a weight of 1 to all its ancestors. Thus, every new block helps in committing the $(f+1)^{th}$ parent.

2. The UCC rule allows us to commit new blocks with a frequency of $O(\delta)$, with every block having a delay of $O(f\delta)$.

In related works such as Sync HotStuff [3], the optimistic latency is $2\Delta + O(\delta)$. They also claim that it is not possible to do better than Δ while tolerating more than $n/3$ faults. We are the first ones to show that we can commit before Δ, by avoiding equivocation detection and employing the UCC rule, and when $\delta \ll \Delta$ or f and n are sufficiently small, i.e., $f < \Delta/\delta$. Our protocol encourages pessimistic choices of Δ as it does not affect the latency of commit.

4 Apollo Protocol

In this section, we discuss the Apollo protocol which uses the UCC rule (Definition 3.1) to build a pipelined, linear SMR protocol for the standard and weak synchrony models.

Proposer sets. We define a proposer set $\mathcal{P}$ consisting of all nodes $\mathcal{N}$. As nodes agree on *misbehavior* from leaders (by committing blocks that contain proof of equivocation/no progress of leaders), we remove the nodes from the proposer set. This allows us to eventually stabilize on a set of leaders of size $n - f$ that are correct and online.

In the weak synchrony model however, some correct nodes can become unresponsive or temporarily experience network partition and thus not be in $\mathcal{O}_t$. Therefore removal of nodes from the proposer set on crash cannot be made permanent as in the case of standard Synchrony. Doing so can make $\mathcal{P} \to \emptyset$. So we place a constraint that the size of the proposer set $|\mathcal{P}|$ must be at-least $n - f$, after which we replace a faulty node with a node that has the highest blocks proposed so far, but not in the proposer set, with ties broken arbitrarily.

4.1 Overview

The protocol proceeds in rounds. In every round, a known leader (derived from $\mathcal{P}$) builds a candidate block extending the highest block known to it. We give a high level overview of Apollo

in Figure 1. Note that the round number and the height of the block need not be the same, since some rounds may not have blocks.

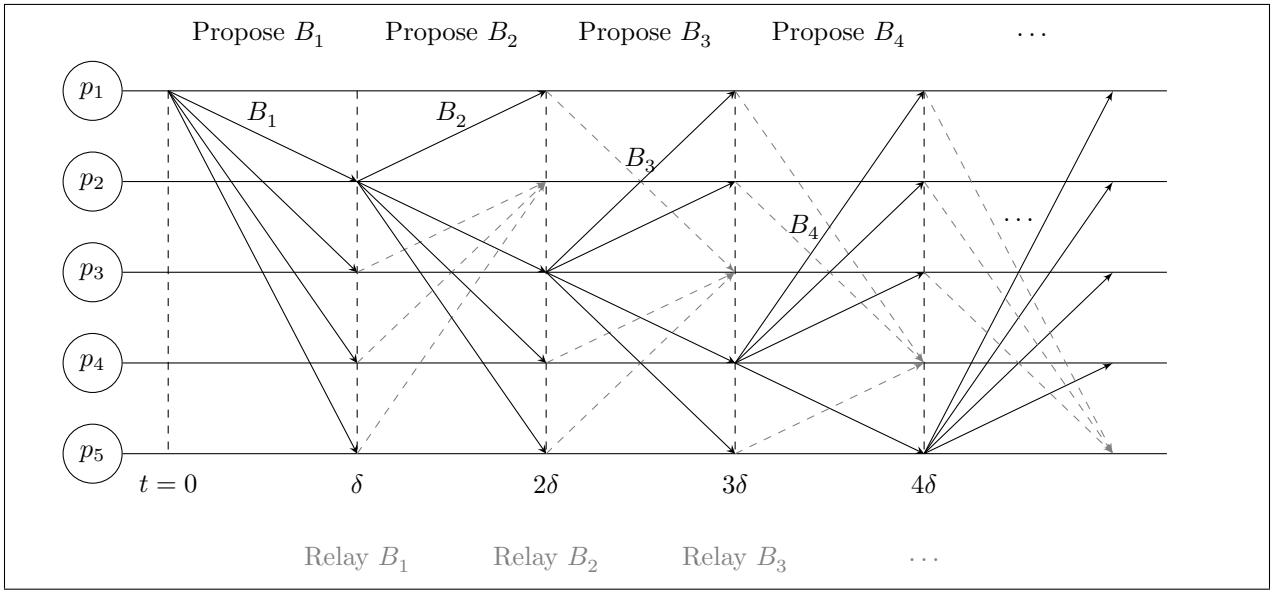


Figure 1: **Overview of the Apollo Protocol in the optimistic case, when all the leaders are correct.** The proposer for round $r + 1$ can immediately propose as soon as it receives the block for round r. Hence, Apollo has a block period of δ, as it does not have to collect votes and certificates for the previous block unlike existing protocols.

Let L_r be the leader of round r.

1. **Propose.** On receiving block B_{k-1} for round $r - 1$, the leader L_r for round r proposes a block $B_k := \langle H(B_{r-1}), \mathsf{cmds} \rangle_L$ by multicasting $\langle \mathsf{propose}, B_k \rangle_{L_r}$ extending the previous block B_{k-1} from the previous leader.
2. **Relay.** On receiving a *valid* proposal for round r, forward it to the next leader L_{r+1}, start timer $\mathsf{blame\text{-}timer}_{r+1}$ to alarm in 2Δ time (refer Figure 3).
3. **(Non-blocking) Commit.** On receiving a *valid* chain of blocks $\mathcal{C} := \{B_0, \ldots, B_\ell\}$ commit blocks $\mathcal{C}[: \ell - f]$ if $\ell > f$.

Figure 2: **Rounds in Apollo protocol**

In Figure 1, the node p_1 is the leader for the first round and performs the *Propose* step. It proposes a block B_1 extending the genesis block since it is the first proposal, but generally nodes extend the block proposed by the leader for the previous round. The node p_2 proposes the next block B_2 immediately after receiving the block B_1. We want to ensure that a correct leader is always able to propose. However, a Byzantine leader can try to slow down the protocol and may not send its proposal to the next leader. To overcome this, all correct nodes forward the proposals of the current round to the next leader. This is shown in Figure 1 as the *Relay step*.

The propose and relay steps keep following each other with different leaders from $\mathcal{P}$. Additionally, in every round, the correct nodes commit blocks after removing the top f blocks from the hash-chain received from the latest proposal. We give technical details in Figure 2.

Let L_r and L_{r+1} be the leaders of round r and $r+1$ respectively.

1. **Blames.** The node p_i, on detecting the following conditions does:
   (a) **No-progress blame.** If $\mathsf{blame\text{-}timer}_r$ has passed and no *valid* block was proposed by leader L_r, then multicast $\langle r, \mathsf{NPBlame}\rangle_i$ along with the latest known local block B_k for round $r-1$. Wait for a blame certificate $\mathcal{C}(r, \mathsf{NPBlame})$. Treat the blame certificate as a virtual block for round r and continue with the relay step (Step 2) of Figure 2. Also multicast the certificate $\mathcal{C}(r, \mathsf{NPBlame})$ to all the nodes.
   (b) **Equivocation.** If there exists two blocks B and $B^\star$ proposed by node $p_j \in \mathcal{N}$ in round $r^\star$ obtained directly or indirectly, multicast a $\langle r, \mathsf{EQBlame}\rangle$ message and the two equivocating blocks B and $B^\star$ with signatures.
2. **Remove leader (optional).** On committing a block with a equivocation blame $\langle r, \mathsf{EQBlame}\rangle_i$ or a certificate of $\mathcal{C}(r, \mathsf{NPBlame})$, remove the leader from the proposer set.

Figure 3: **Handling Byzantine behaviour in Apollo protocol**

4.2 Handling Byzantine Faults

In this section, we describe and give an overview of Apollo during equivocation and crashes of nodes. A concise technical description is presented in Figure 3.

Block equivocation. A leader can equivocate by sending different blocks to different correct nodes. As stated earlier, unlike existing synchronous SMR protocols [1, 3], Apollo does not need to detect equivocation to preserve *safety* or *liveness*. Consider a leader L_r equivocating in round r. At least one of the blocks reaches the next leader L_{r+1} through the *Relay* step. It will immediately propose the next block. In general, an equivocation is detected by correct nodes in two ways:

1. A correct node whose head of the chain is $B_{k'}$ may obtain a block B_k from some leader L_r with $k' < k$ and an unknown parent hash. It will immediately request all the blocks $B_{k'}, \ldots, B_{k-1}$ until B_k connects to the node's local chain. If L_r cannot provide valid ancestors within 2Δ, then the correct node blames L_r. A correct L_r can always respond to such queries and therefore not get blamed by correct nodes. When the parent block is received, a correct node may realize equivocations due to contradiction with the local chain.

2. A correct node can get two different blocks during the *Relay* step (Step 2) of the protocol.

In both of these cases, the correct nodes multicast the proofs of equivocation to all the other nodes if it detected the equivocation directly or via equivocation blame from others. Note that this is basically a reliable broadcast of equivocation blame. All correct nodes include the equivocation blame as a meta-transaction in their future proposals until it is committed.

It is important to not act on the equivocation until it is committed through some block. This is because we do not use timing guarantees and we do not rely on Δ to ensure that all nodes have detected and agreed on the equivocation. Instead, we rely on the fact that sufficient correct nodes have extended the block containing the equivocation blame and therefore all other correct nodes must know that the node has equivocated by then (or will know once the network partition clears or the unresponsive nodes become prompt).

Crashed leaders. We reiterate that we always want a correct leader to be always able to propose blocks. Consider leaders L_r and L_{r+1} for rounds r and $r+1$ respectively. Let L_r not propose any block to any node. Now, the correct nodes at this point in time, could be processing/waiting for

blocks at different rounds $\leq r-1$. The first correct node to finish processing the block B_k for round $r-1$, will wait for 2Δ time (we will describe soon why to wait for 2Δ) after relaying B_k before blaming L_r. Upon timing out, a correct node cannot be sure if all the correct nodes are waiting for a proposal for round r, since our protocol can proceed at network speed for some nodes but slowly for other correct nodes (i.e., messages are delivered toward the end of the Δ delay). Therefore, some correct nodes can be waiting for a block for round r, whereas others may be waiting for blocks at rounds $r' < r$. This case can also occur if Byzantine leaders send the proposals to some correct nodes, who will then be ahead in round number when compared to the nodes that did not receive the proposals.

In any case, upon timing out, a correct node on time out for round r, sends the latest block B_k to all the correct nodes in order to synchronize the round for all nodes to $r-1$. We do this multicast only after a time out and not for every round in order to obtain the desired linearity in the steady state as this step has a communication complexity of $O(n^2)$ if $n-f$ nodes time out. We know that within 2Δ all nodes will relay the proposal to L_r and then blame when it does not respond. On collecting $f+1$ such blame messages, all the correct nodes build a virtual block for round r consisting of blames against L_r. Using threshold signatures, this block has $O(1)$ size. From this point, we continue with the relay (Step 2) of this virtual block to L_{r+1} just as though we received a proposal with this virtual block from L_r.

Why multicast blames? We always multicast blame certificates/ equivocations to all the other nodes. We do this because the leader L_r of a round r could be Byzantine and can trigger a time out for some correct nodes for round r but later produce a valid block B_k for round r. By multicasting the blame certificates/equivocations to all the other nodes, the correct nodes will include the blame certificate as a meta transaction, which will eventually be committed thereby ensuring that the leader L_r cannot cause such delays in the future.

Blame timers. Apollo using UCC Rule allows us to use *round-relative* timers, i.e., blame based on the latest round. Unlike Sync HotStuff [3], which uses *view-relative* timers, where in a view v, the condition for triggering a no-progress blame is to not receive p blocks in $(2p+4)\Delta$ time. Assume that the first 1000 proposals are made at network speed after which the leader crashes. In Sync HotStuff, the nodes needlessly wait for 2004Δ before blaming the leader. We overcome this, since our timers are always rooted to the last received block.

Why 2Δ timeout is sufficient? Again, we want to ensure that a correct leader is always able to propose and is not incorrectly blamed. Every node forwards the previous block to the current leader. This will take Δ time in the worst case. A correct leader will then immediately propose since it has a block to extend. It will take another Δ for this block to reach the correct node in the worst case. Therefore waiting for a total of 2Δ after relaying the block is always sufficient for a correct node to ensure that it does not blame a correct leader.

4.3 Security Analysis in Standard Synchrony

In this section, we prove the safety of Apollo protocol in standard synchrony. The commit rule employed by Apollo from Definition 3.1 guarantees that no two conflicting blocks will be committed by any correct node. On a high-level, we want the following important properties to ensure security:

1. Correct leaders are always able to propose.
2. Correct proposals are always committed.

Valid proposals. In the commit rule in Figure 2, we mention *valid* proposals. The definition of *valid* proposals vary with the network assumption of standard or weak synchrony.

Definition 4.1 (Valid proposal)**.** *A valid proposal for round r consists of a block B_k extending the parent B_{k-1} from round $r' < r$, and contains blame certificates $\mathcal{C}(r'', \mathsf{NPBlame})$ for $r' < r'' < r$.*

For the standard synchrony assumption, we add an additional constraint to chain validity (apart from the definitions from Section 2.3):

Definition 4.2 (Valid chain for standard synchrony)**.** *A valid chain $\mathcal{C} := \{B_0, \ldots, B_\ell\}$ of size ℓ consists of:*

- *Hash chain of blocks (from Section 2.3), i.e., for any $i \in [0, \ell)$, B_{i-1} is the parent of B_i, or*
- *A block B_k extending the parent block B_j with valid blame certificates $\mathcal{C}(i, \mathsf{NPBlame})$ or of the type $\mathcal{C}(i, \langle r, \mathsf{EQBlame}\rangle)$ with equivocating blocks, as virtual blocks for round i, where $i \in (j, k)$, $j \in [1, k)$, and $k \in [1, \ell]$.*

Definition 4.2 defines a valid chain to be a chain consisting of valid blocks including blame certificates serving as virtual blocks. The standard synchrony allows this as the assumption dictates that in the lifetime of the protocol only f nodes can crash/equivocate. From the tamper resistance property (Lemma 2.1), we are guaranteed that a Byzantine node cannot go back in time and replace older proposals with virtual blocks, thus ensuring the protocol safety.

We first prove that a block proposed by a correct node will always be committed.

Theorem 4.1 (Correct commit for correct leaders)**.** *For any round $r \geq 1$, if the leader L_r is correct and proposes a block B_k, then B_k will be committed by all correct nodes in round $r + f + 1$.*

Proof. Safety. To understand why this is true, let us take a look at what a Byzantine leader L_{r+1} of round $r + 1$ can do with B_k. The Byzantine leader can decide to not extend the block. The only way to do this is to obtain $f + 1$ blames against the correct leader. This is not possible for a correct leader. Let p_i be the earliest node to reach round r, either via a time out from round $r - 1$ and obtaining a blame certificate, or by obtaining a block B_{k-1} for round $r - 1$. Let t be the global clock time at this point. Before time t, all correct nodes are waiting for blocks for rounds $r' < r$. Now, p_i forwards its block (virtual or real) for round $r - 1$ at time t. The leader L_r being correct, will respond to all the nodes with a block B_k for round r by time $t + 2\Delta$. Therefore, no correct node will time out for L_r, and L_r can always respond to block requests from correct nodes.

Liveness. A Byzantine leader L_{r+1} can extend B using two proposals B_{k+1} and $B^\star_{k+1}$. In this case, eventually some version of the chain with one of the two proposals B_{k+1} or $B^\star_{k+1}$, will be $f + 1$ long, when the $(f + 1)^{\text{th}}$ leader proposes a block. This ensures that B is committed. If f consecutive leaders decide not to extend B, then after obtaining blame certificates for the f Byzantine leaders, the next leader will extend B_k and thus commit B_k by $f + 1$ rounds. □

Now, we prove the safety of the commit rule.

Theorem 4.2 (Commit safety)**.** *For any height $k \geq 0$, if two correct nodes commit to blocks B and $B^\star$, then $B = B^\star$.*

Proof. For $k = 0$, the proof is trivial since the genesis block B_0 is agreed upon by assumption.

Assume that two correct nodes commit to two blocks B_k and $B^\star_k$ with $B^\star_k \neq B^\star_k$ for some height k. This implies that there exist two valid (refer Definition 4.2) chains $\{B_k, \ldots, B_{k+f}\}$ and $\{B^\star_k, \ldots, B^\star_{k+f}\}$. From the tamper resistance property (Lemma 2.1), this implies that there are two

proposals for *all* blocks starting from height k to $k+f$. This implies $f+1$ nodes equivocated or have blame certificates. This is a violation of the assumption that only f nodes in the system are Byzantine. Therefore, we cannot obtain two chains as described previously. □

To prove liveness, from Theorem 4.1, we know that in one of the nodes in rounds $k^\star \in \{k, \ldots, k+f\}$ is correct, and therefore is eventually (by round $k^\star + f + 1$) committed, thereby committing B. We prove it formally in Theorem 4.3.

Theorem 4.3 (Apollo liveness)**.** *Assuming standard synchrony, the Apollo protocol always makes progress, and commits blocks with a period of at most* 6Δ*.*

Proof. Because the clocks of different nodes might be different, we use a global clock time unknown to the nodes but only for external observers observing the protocol to measure the difference in time. Let t be the current global clock time, and let the first correct node be reach round r at this time, by obtaining a blame certificate or a block for round $r-1$. In the worst case, due to network delay or due to Byzantine nodes, a correct node will blame a crashed leader L_r for round r at time $t+2\Delta$, followed by sending the block to all the nodes, which will reach all the correct nodes in Δ at time $t+3\Delta$. After this step, all the correct nodes will reach round r. Now all these correct nodes forward the block for round $r-1$ to the crashed leader again and wait for 2Δ before sending a blame at time $t+5\Delta$. This blame takes an additional Δ time to reach all other correct nodes at time $t+6\Delta$. At this point, we commit one block, because this virtual block extends the local chain of every correct node. Therefore in the worst case, it can take 6Δ for a block to be committed, but we will always make progress. The 6Δ wait only occurs f times in the lifetime of networks assuming standard synchrony. If all the nodes are correct, but the network is slow, then we progress with a period of Δ always. □

Theorem 4.2 and Theorem 4.3 prove that Apollo is a secure SMR protocol under the standard synchrony assumption.

4.4 Security Analysis in Weak Synchrony

In this section, we prove the safety of Apollo protocol in weak synchrony. A constraint in the weak synchrony model is that nodes can be offline. The key difference in the proof, lies in the definition of *valid* chain. In this model, the network can adaptively go down so that there can be indefinitely long chains of blame certificates, and therefore we cannot safely treat blame certificates as virtual blocks and help commit blocks.

Consider the following scenario. The nodes who are leaders in every round become offline in every round, i.e., $L_r \notin \mathcal{O}_t$ for every round at the corresponding global clock time t. This can continue indefinitely, and the protocol can never make any progress. This is not just a limitation of Apollo, but of all the existing SMR protocols [3, 14] in this model.

The above limitation also implies that we need to revise our definition of valid blocks as we can no longer treat a blame certificate as a virtual block at that height. We give the revised definition of valid chain in Definition 4.3.

Definition 4.3 (Valid chain for weak synchrony)**.** *A valid chain* $\mathcal{C} := \{B_0, \ldots, B_\ell\}$ *of size* ℓ *consists of a hash chain of blocks* B_i *of round* r*, where* $i \in (0, \ell]$*, extending the parent block* B_{i-1} *of round* $r' < r$ *with* B_i *containing blame certificates or equivocation blames with equivocating blocks for every round* $j \in (r', r)$*.*

We first observe that if the proposer for round r, obtains a block for round $r-1$, and is online, then its proposal cannot get a blame certificate.

Lemma 4.4. *Let $p_i \in \mathcal{N}$, be the proposer for round r. Let it obtain a block from round $r-1$ at time t. If p_i is online at time t and proposes B, then B cannot obtain a blame certificate.*

Proof. In the time interval $t' \in (t, t+\Delta]$, $\mathcal{O}_{t'}$ can change at most once (by model definition). But the size of $\mathcal{O}_{t'}$ is $\lfloor \frac{n+1}{2} \rfloor$ which is at least $f+1$, and therefore some $f+1$ correct and online nodes will hear this proposal, and thus never blame p_i. This ensures that B, and thus a proposal for round r, can never obtain a blame certificate. □

Theorem 4.5 (Commit safety for weak synchrony)**.** *For any height $k \geq 0$, if two correct nodes commit to blocks B and $B^\star$, then $B = B^\star$.*

Proof. Let p_i, $p_j \in \mathcal{N}$, be two nodes that commit to blocks B_k and $B_k^\star$ at height k. For the weak synchrony model, this implies that there exist two chains $\mathcal{C} := \{B_k, \ldots, B_{k+f+1}\}$ and $\mathcal{C}^\star := \{B_k^\star, \ldots, B_{k+f+1}^\star\}$ through which p_i and p_j commit to B_k and $B_k^\star$ respectively.

Assume $B_k \neq B_k^\star$. This implies $B_{k+1} \neq B_{k+1}^\star, \ldots, B_{k+f+1} \neq B_{k+f+1}^\star$ (from Lemma 2.1) and since a block cannot have two parents. From Definition 4.3, we also know that no block B_i and $B_i^\star$ can be virtual blocks, i.e., are blame certificates or equivocation blames, for $i \in \{k, \ldots, k+f+1\}$.

If all the blocks at height $k \leq i \leq k+f+1$ are from the same rounds, then $f+1$ nodes have equivocated, and are therefore Byzantine, which contradicts our assumption that there are only f Byzantine nodes in the system.

Assume that some $b \leq f$ nodes are Byzantine. Now, there are $d = f - b$ *correct* nodes, that can be offline at any time t.

Let us start from the block at height j with $j = k$ initially with $d = f$ and $c = 0$. We show that by the time $j = k+f+1$, the chain grows $f+1$ long, $B_k = B_k^\star$.

If the proposer for B_j is a Byzantine node, increment $j \leftarrow j+1$, $d \leftarrow d-1$, $c \leftarrow c+1$ and repeat the analysis.

If the proposer for height j is a correct and online node, then from Lemma 4.4, all future online nodes obtain B_j, and the round can never obtain a blame certificate. Therefore, Byzantine nodes cannot extend or equivocate blocks after height j. The analysis terminates with $B_j = B_j^\star$ which results in $B_k = B_k^\star$.

If the proposer for height j is an offline node, then the online nodes will build a blame (from the *No-progress Blame* step in Figure 3), and continue with the next proposer for height $j+1$. The blame step results in all of these nodes sending their latest blocks, again ensuring that all the correct and online nodes extend the same block for height $j-1$, leading to $B_k = B_k^\star$.

Now, an offline proposer can propose a block at height j, which is delivered to another offline proposer for height $j+1$, which can propose a block for height $j+1$. However, this series of steps can only occur up to $0 \leq d \leq f$ rounds, eventually leading to one of the above scenarios. Whenever, a node becomes online, it receives messages from all previous online nodes, leading to a synchronization of chains which ensures that this correct node will not extend another block at height $j-1$ and thus $B_k = B_k^\star$.

Therefore, by the time $j = k+f+1$, we have ensured that one of the above scenarios will be encountered, all of which lead to: if any correct node observes a block at height $k+f+1$, then $B_k = B_k^\star$. □

Theorem 4.6 (Apollo liveness for weak synchrony with strong — GST assumption)**.** *Apollo protocol achieves liveness in weak synchrony assuming the existence of a strong - Global Stabilization Time* GST_s.

Proof. The protocol can stop making progress, when the network keeps making the leaders become unresponsive, leading to long chains of blame certificates. After time $t = \mathsf{GST}_s$, this stops happening, and the protocol makes progress, and commits blocks. □

Theorem 4.7 (Apollo liveness for weak synchrony with weak - GST assumption)**.** *Apollo protocol achieves liveness in weak synchrony assuming the existence of a weak - Global Stabilization Time* GST_w*.*

Proof. We must use the proposer sets in order to achieve liveness under the weak - GST assumption. After some time $t = \mathsf{GST}_w$, when there are $n - f$ nodes that are persistently online, the proposer set $\mathcal{P}$ stabilizes to these $n - f$ nodes, and Apollo enjoys liveness. □

4.5 Discussion

To the best of our knowledge, we are the first to deviate from the philosophy of detecting equivocation in a $n > 2f$ setting by using sufficient number of proposals to commit a block. In this section, we explore how this work can be observed through the lens of some classic protocols.

Parallels to Dolev-Strong. The UCC rule can also be viewed as pipelined, linearized and hash-chained version of the original Dolev-Strong protocol [16]. In the classic Dolev-Strong broadcast protocol (which tolerates $f < n$), a node accepts a value in round $1 \leq i \leq f + 1$, if there are i signatures (including the sender's) for a value. And finally in round $f + 1$, it accepts a value if there is only value accepted so far.

In UCC, our rounds $0 \leq i \leq \infty$, go on forever. On a high level, for the standard synchrony version, a block B is committed/accepted in round i if there are $f + 1$ implicit votes on it, where each vote is in the form of an extension of the block or blame certificate, which occurs naturally for all blocks proposed before round $i - f - 1$. Our pipeline ensures that the Dolev-Strong rule can be pipelined for multiple broadcast instances since every block extension counts as support for all its parents.

Parallels to Phase-King. The Phase-King algorithm [5] is another classic protocol which uses a gradecast protocol to achieve consensus in $f+1$ rounds by introducing *King consistency*, a property where if the leader or king of a round is correct, then all the nodes output the same value and have a grade $g = 1$ along with the standard *validity* property (to recall, it states that if all the nodes have the same input v, then all the correct nodes output v). Our UCC rule can be viewed as a pipelined application of *Phase King* [5] applied per block.

For standard synchrony, our UCC rule, also has a similar property, where if the leader of a round r is correct a proposes v for height k, then the block is *pinned*, i.e., no node will output a different v' for height k. Just like the Phase-King algorithm, the nodes cannot detect when this occurs, and therefore must continue on till $f + 1$ rounds after which they are sure that there must have a correct king/leader.

Another reason why we draw attention to classical protocols is because classical results also apply to the UCC rule. For instance, using random beacon or OLE (oblivious leader election) it is shown how to achieve expected constant round agreement [23, 18], by simply choosing the king randomly. The same results can also be translated to UCC.

Parallels to Casper [10]. In our work, we treat the $f + 1$ suffix as a certificate analogous to super-majority votes from stakeholders in PoS protocols [10, 19]. Casper [10] uses the rule that if $2f + 1$ stake votes on a branch of the chain, then the prefix of the such a chain is safe to commit. The UCC rule can be viewed as generalization of Casper, where every one of the $2f+1$ explicit vote

is converted into an implicit vote by extending the chain, thereby contributing to the throughput by doing useful work.

On hash-chains. Another question that arises is whether we can remove hash-chains. The UCC rule requires tamper-resistance (refer Lemma 2.1). Removing the hash-links with simple counters do not work and violates safety. However, the chaining need not be explicit or consists of hashes as we have described. The chaining can be made implicit by including the signature of the previous block as input before signing the current block.

5 Head Synchronization

We observe that any particular block always incurs a commit latency of $O(f\delta)$, even in optimistic conditions. Depending on the choice of Δ, urgency of some transactions, for sufficiently large n and therefore f, or at regular intervals, we may wish to synchronize the heads of all the correct nodes. In other words, we want all the nodes to catch up to the head of the chain. We give a synchronization protocol with this exact goal: To synchronize the heads of chains of all the correct nodes.

Requirement. During synchronization, we can potentially have correct nodes with different heads at different heights. This happens because different nodes see the synchronization certificate at different heights, or different forks due to equivocation by the previous leaders. At the end of synchronization, we want all the nodes to commit to the head of the chain, so that the nodes accept new blocks from new leaders without breaking the hash-chain.

Sync certificate. Depending on the use case, we make the nodes sign a message $\langle \texttt{Sync}, v \rangle$. Interested parties (clients or nodes) can collect $\langle \texttt{Sync}, v \rangle$ from at least $f + 1$ nodes to build a *synchronization certificate* $\mathcal{C}(\texttt{Sync}, v)$. This can be done by nodes periodically at fixed wall clock times (such as 12 AM everyday), at fixed block intervals (every 1000 blocks), or with the assistance of a client such as for high priority transactions with high fees, or based on an internal client hierarchy (some clients may be special and more important). After obtaining the sync certificate, it is sent to a correct node to start the synchronization protocol.

Safety. At the end of the synchronization protocol, we want to ensure that all correct nodes agree on the same height, and start accepting blocks from the same leader so that Apollo protocol can resume correctly. The latter can be solved by simply resetting $\mathcal{P}$. Theoretically, the former requires running a Byzantine Agreement (BA) protocol (which we recall in Definition 5.1) among all the nodes, whose *validity* condition guarantees that if all the correct nodes start with the same input B_{head}, then all correct nodes output the same block; otherwise a consistent block is output by all correct nodes.

Definition 5.1 (Byzantine Agreement)**.** *Let the set $\mathcal{N} := \{p_1, \ldots, p_n\}$, be the nodes in the system, f of which are Byzantine. Each correct node has an input v_i. A protocol Π_{BA} is said to be a BA protocol if it satisfies the following conditions:*

(i) ***Safety.*** *If any two correct nodes $p_i, p_j \in \mathcal{N}$ output out_i and out_j respectively, then $out_i = out_j$.*

(ii) ***Validity.*** *If all correct nodes start with the same value v, then for any correct node p_i, $out_i = v$.*

(iii) ***Termination.*** *All correct nodes $p_i \in \mathcal{N}$ eventually output a value out_i.*

The above *validity* requirement alone is sufficient in not sufficient to ensure safety for the weak synchrony setting. Some nodes could be sluggish while the other nodes are synchronizing. A sluggish node may become prompt after synchronization, and may accept incorrect blocks in the wrong view. To resolve this, we certify the first $f+1$ proposals in every view. This ensures that the head after synchronizing cannot get $f+1$ proposals without being certified and therefore a sluggish node cannot commit stale blocks beyond the head.

Let Π_{BA} be a black-box BA protocol that satisfies Definition 5.1 when $n > 2f$. Let v be the current view.

1) **Relay sync certificate.** On receiving a synchronization certificate $\mathcal{C}(\texttt{Sync}, v)$ directly, or via multicast from another node, multicast the message $\langle \mathcal{C}(\texttt{Sync}, v), B_{head}, B_c \rangle$ to all the other nodes with the latest head block B_{head} and highest committed block B_c. Wait for 2Δ and stop accepting new proposals. Update head blocks and committed blocks if longer chains are received from others.
2) **Byzantine agreement.** Run the protocol Π_{BA} with blocks $\{B_c, \ldots, B_{head}\}$ as input.
3) **Resume.** When the BA protocol outputs $B^\star \leftarrow \Pi_{BA}$, send vote for $B^\star$ and send it to the first leader after resetting $\mathcal{P}$.
4) **New view validate.** In the new view $v+1$, the first $f+1$ proposals are only valid if they are certified, i.e., block B_r contains $\mathcal{C}(B_{r-1})$. This entails that during the proposal step for the first $f+1$ blocks in a view, the proposer first sends a block, and the remaining nodes on verifying the correctness of the block, send a vote to the proposer, who finally proposes the certified block.

Figure 4: **Description of a generic synchronization protocol** for Apollo using a black-box BA protocol Π_{BA}.

View. Similar to some SMR protocols [11, 3, 9], we use a counter called *view* to denote a view number. In PBFT [11] and Sync HotStuff [3], a view represents blocks proposed by a single leader, and a view change corresponds to changing a leader. In Apollo protocol, we look at views as proposals from an imaginary virtual leader (in the head), and a view change as changing this virtual leader. We observe that view change protocols are Byzantine Agreement protocols that smoothly transition into steady state protocol with the next leader.

Limitations and bounds. Since view change protocols are a form of BA, and we require BA, our synchronization protocol is also bound by the limitations of BA. A pessimistic lower bound by Dolev & Strong [16] states that for synchronous networks, agreement tolerating f Byzantine nodes, requires at least $f+1$ rounds to reach agreement in the worst case. This is true for view change protocols as well when f continuous leaders are Byzantine, they can trigger f consecutive view changes. We are also bound by the same bound and we can hope for early termination only when the Byzantine nodes follow the synchronization protocol, otherwise the synchronization protocol will also take $\geq f$ rounds.

Since we make the connection that a synchronization is the same as a view change in Sync HotStuff [3] and all we need is to achieve efficient Byzantine Agreement. We assume the existence of a black-box BA protocol Π_{BA}, and show how to construct a secure Synchronization protocol for Apollo in Figure 4.

Security Analysis for Synchronization

Theorem 5.1 (Secure synchronization)**.** *If Π_{BA} is a secure BA protocol in the standard or weak synchrony model, then the synchronization protocol defined in Figure 4 guarantees all the properties from Definition 2.1.*

Proof. Since Π_{BA} is a secure BA protocol, we know that all correct nodes output and commit to the same chain $\mathcal{C}$. Therefore, all correct nodes commit to the same chain $\mathcal{C}$ after synchronization for standard synchrony.

However, in the weak synchrony model, let some correct nodes be unresponsive before, and throughout the synchronization protocol. Now, Byzantine nodes can trick the nodes into accepting incorrect chains by emulating messages as though no synchronization occurred. In this case, the unresponsive nodes can commit blocks only up to the last block in $\mathcal{C}$. The Byzantine nodes can keep adding equivocated blocks extending $\mathcal{C}$ up to $f-1$ blocks, since at least one correct node is unresponsive. After which, all the blocks contain the synchronization certificate, which ensures that all correct nodes move to the synchronized chain. □

6 Performance Analysis and Experiments

Setup. We ran all of our experiments on AWS c5.4x-large instances. These machines have 16 vCPUs, 32 GB RAM, 8 GB of storage. The instances are deployed in a Virtual Private Cloud, a separate subnet for the protocol nodes. AWS advertises 10 Gigabit networking for these instances.

We implement both Sync HotStuff and Apollo in Rust [6]. We perform several optimizations to both Sync HotStuff as well as Apollo. Interested readers can refer to Appendix B for the details.

Transactions. In our implementation, a transaction $tx := (\mathsf{data}, \mathsf{request})$ consists of a vector of bytes data and request. data is a serialized 8-byte unsigned integer serving as the transaction ID. The request field is the payload of the transaction, containing data for the transaction such as data for a write request. In our experiments, we use the size parameter p to denote the size of request sent by the client.

Block. In our implementation, we use a block format that is common to both Sync HotStuff and Apollo, and separate the protocol specific fields into the proposal message. A block B contains a header $header := \left\langle h_{r'}, L, r, \vec{\mathcal{C}}(\mathsf{Blame}), \vec{tx}, \vec{resp} \right\rangle$, , where $r' < r$, $h_{r'}$ is the hash of the parent block of round r', r is the round where L proposed the block, and $\vec{\mathcal{C}}$ is a vector of blame certificates from round r' till the height r, $\vec{tx}$ is a vector containing the hashes of the transactions included in the block, and $\vec{resp}$ is filled with response data of size p for every transaction when sending the blocks to the clients.

Clients. We use two types of clients: the first kind, used in Sync HotStuff and HotStuff, waits for $f+1$ acknowledgements from the server to be sure that it is safe to commit a block. This is necessary for safety, as the block on its own does not contain any information that proves finality. The second kind is specific to Apollo. Observe that in our protocol, the UCC rule can be applied by anyone in the system, including the clients. Therefore, in Apollo, the proposer simply acknowledges blocks to all its connected clients, and the clients can finalize blocks using the same rule. Considering light-weight clients, in Apollo a single node can provide a block and the signed headers for the next $f+1$ blocks and convince the client correctly that the state in the block is final. We take advantage of this feature in the client for Apollo.

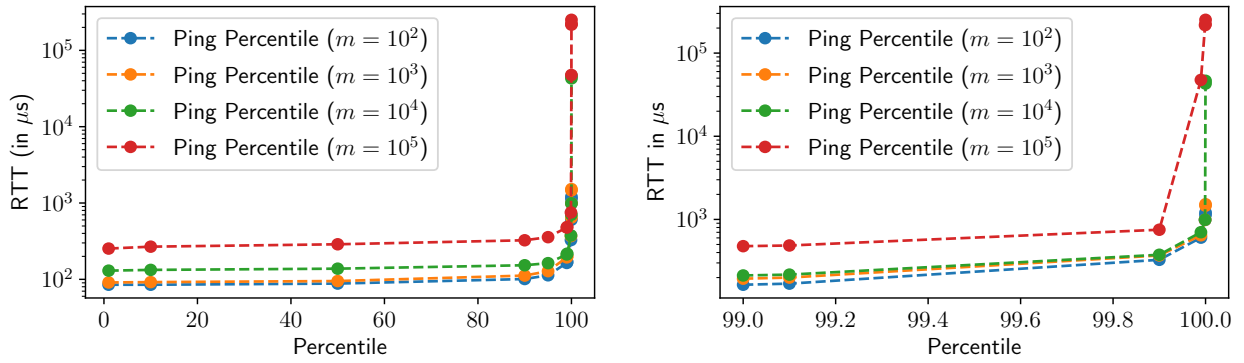
Measurement details. Our implementation of the client uses a window parameter w called the work load analogous to the public implementation for Sync HotStuff; it always maintains w outstanding commands. As soon as some blocks are committed, fresh commands are issued to maintain w outstanding commands. Upon receiving a transaction, a node adds it to its pending transaction buffer. The leader starts the proposal when sufficient transactions (sufficient to build a block with b transactions) are received.

In Sync HotStuff, when a block is committed, the nodes acknowledge the block by forwarding the block to all the connected clients. The client on receiving $f + 1$ such block acknowledgements, records the time and computes the latency for the transaction. In Apollo, the proposer for a round also proposes the block to its clients, and the clients apply the UCC rule independently to commit blocks.

6.1 Latency estimation

In order to estimate the practical values for δ and Δ and to observe the relationship between δ and Δ, we run the following experiment.

Experiment. We build *ping-client* and *echo* nodes in our implementation. The ping-client node streams a count of c messages of size m in bytes to an echo node. The echo node simply echos the messages it receives back to the sender. The ping-client then measures the round trip time for every message, and outputs it at the end of the experiment.



(a) In this experiment, we send $c = 10^6$ ping messages (containing fixed data) of size m to an echo server, and measure the round trip time (RTT) for every ping.

(b) In this figure, we take a deeper look at the last percentile from Figure 5a.

Figure 5: **Latency distribution between two nodes in AWS networks.**

By sending pings continuously to an echo server and measuring the round trip times, from Figure 5a, we can clearly see that δ is actually very small (by 3 orders) most of the time. From Figure 5b however, weobserve that the ping time shoots up unpredictably. To assume bounded synchrony, we have to assume the worst of these values as Δ, and that implies setting $\Delta \gg \delta$. The weak synchrony model allows such unexpected Δ violations as long as these are limited to a small set of nodes and for a sufficiently small duration of time (as long as we can buffer).

As a takeaway from this experiment and also as cautioned in Sync HotStuff [3], we can draw the conclusion that for practical synchronous SMR protocols, we require large values of Δ, thereby motivating protocols that avoid the use of Δ in its commit rule.

For our experiments, we use the value of Δ for our protocols as $50ms$ in conjunction with the value used in the original Sync HotStuff paper, so that we can compare and contrast our results with numbers from their paper.

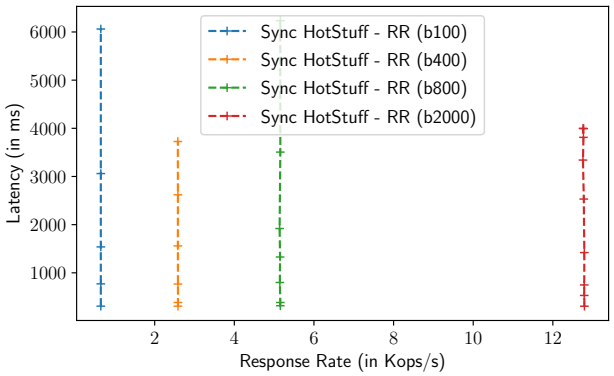


Figure 6: **Offered load** (w) **vs. Response rate (Kops/s) for Sync HotStuff - Round Robin variant** for various block sizes. The latency shoots as soon as the optimal point for w crosses $b/3\Delta$.

6.2 Synchronous round robin protocols

In this section, we evaluate round robin protocols in general. We make simplifying assumptions to Sync HotStuff's view change protocol and implement a round robin variant of Sync HotStuff with every round consisting of 3Δ. Note that this is still lower than some of the other round based protocols mentioned in Table 1. We present the response rate vs. latency in Figure 6. We observe that Sync HotStuff RR can produce $\frac{1}{3\Delta}$ blocks every second as its response rate. Therefore, in order to achieve a response rate of, say 100 Kops/s, we need a block size b of approximately 15K. We can clearly conclude that round robin protocols that rely on Δ bounds, need bigger block sizes, which leads to larger wait times for clients to get their transactions committed.

Jumping ahead, in Figure 7a, we observe that Apollo, despite being a round robin protocol, initially lags behind Sync HotStuff stable leader protocol because of the overheads induced by the round robin structure. The proposers of a round robin protocol are not always ready to immediately propose since they may not yet be aware that it is their turn to propose, whereas in a stable leader protocol, there is practically no lag as the leader can immediately propose after finishing with the previous proposal and obtaining the certificate. However, as the block size increases due to UCC being independent of Δ, we observe that with increasing block sizes, this gap is quickly closed by Apollo, thereby demonstrating the efficiency of the commit rule.

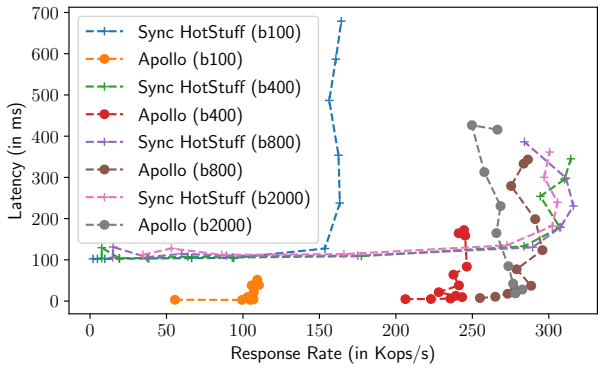
6.3 Performance Evaluation

Basic performance. In this set of experiments, we measure the performance of Apollo and Sync HotStuff with varying system parameters.

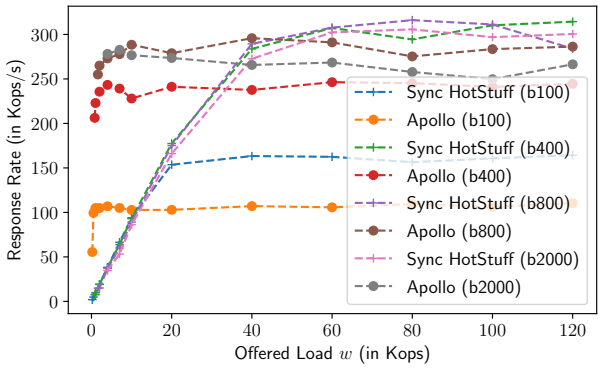
Experiment. We run both variants of Sync HotStuff, and Apollo using $n = 3$ instances. We denote the block size b as the number of transactions in a block. We use one client that maintains w pending commands and waits until $1,000,000$ transactions are committed, after which the client reports the response rate (total number of transactions committed/total time) and the average latency of transactions (time between sending and committing).

Figure 7a, Figure 7b, and Figure 7c show our results for various block sizes. We would also like to note that these numbers are better than the numbers reported for Sync HotStuff in their original paper.

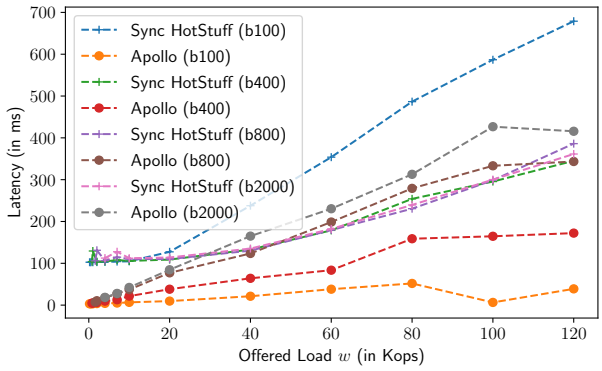
We also observe that the latency for Sync HotStuff is constant for most response rates, until after some point, the system quickly saturates and the latency shoots up. But looking plainly at



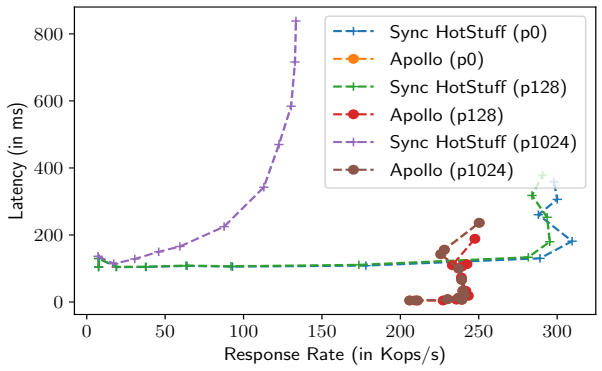
(a) **Response rate (Kops/s) vs. Latency** for various block sizes.



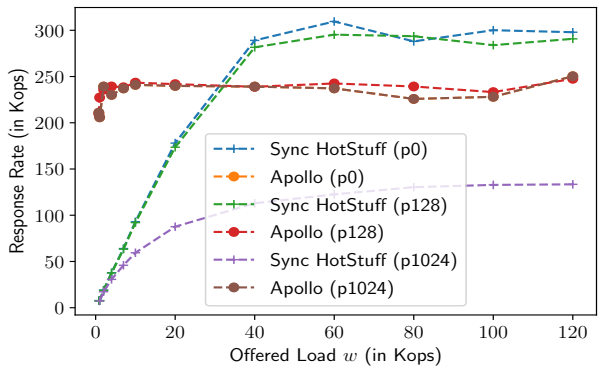
(b) **Offered load** (w) **vs. Response rate (Kops/s)** for various block sizes.



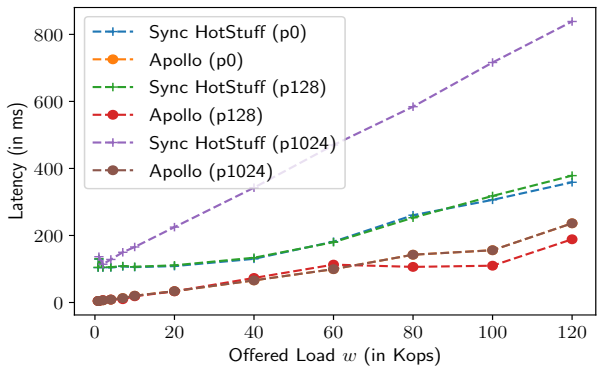
(c) **Offered load** (w) **vs. Latency** for various block sizes.



(d) **Response rate (Kops/s) vs. Latency** for various payloads.



(e) **Offered load** (w) **vs. Response rate (Kops/s)** for various payloads.



(f) **Offered load** (w) **vs. Latency** for various payloads.

Figure 7: **Basic performance statistics for Apollo and Sync HotStuff** with system parameters: system size $n = 3$, delay $\Delta = 50$ms, payload $p = 0$ and block size $b = 400$.

Figure 7a for Apollo, one might be tempted to draw the same conclusion for Apollo. This is not the case, and we show the variation of these two metrics, i.e., response rate and latency, with respect to the third parameter w in Figure 7b and Figure 7c. In Figure 7b, we can clearly see that with the current experiment design used in evaluating Sync HotStuff, Apollo always manages to extract the

maximum throughput from the clients, irrespective of w. In Figure 7c, we can see the benefit of using a commit rule that is independent of Δ. Under optimistic conditions, we can clearly observe latencies of about $3-5$ ms $\ll \Delta$, which was typically only enjoyed by asynchronous protocols, until now. At this scale, with $n = 3$, this is almost the same as HotStuff using the 2-chain rule without the signatures but with higher fault tolerance.

Payload. In this set of experiments, we measure the impact of the size of payloads on consensus.

Experiment. We use the same setup used previously, except we fix the block size to $b = 400$. This block size is not an advantage for Apollo, but we choose the same because this allows us to compare the work with existing literature. This experiment emulates real world SMR protocols where the actual transaction could be read/write request, and therefore apart from responding to the client with the committed transactions, a node may also need to provide a client with the response after committing the transaction, and similarly, the client may include payloads as a part of its transaction. At this point, we stop comparing ourselves with Sync HotStuff-rotating leader variant, as it is too slow.

In Figure 7d, we present the response rate vs. latency curves for Sync HotStuff: stable leader variant and Apollo. We can clearly see that at lower payloads Sync HotStuff enjoys better performance than Apollo consistent with the results from Figure 7a, but as the payload increases we see a drop in the performance of Sync HotStuff. This is because a Sync HotStuff client requires $f + 1$ nodes in the system to provide the payload to the client, resulting in a bottleneck if the payload is too large, as the client socket is always busy writing. This results in increased latency as measured by the client, which also drops the response rate of the system.

However, Apollo being a round-robin protocol employing the UCC rule means that only the proposer will be sending the payload to all its connected clients, and the node has $n - 1$ turns before the next time the socket is needed. Similarly, the client will also be receiving proposals from multiple sockets at the same time, and with our efficient work stealing concurrency, this means parallelism for the clients as well. Therefore, we observe almost no loss in response rate and latency in Apollo as compared to the $p = 0$ case.

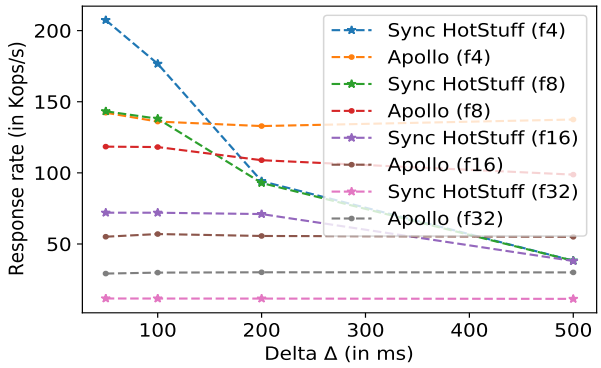
Once again, Figure 7e shows that Apollo once again manages to extract the maximum response rate from the system, and Figure 7f shows how we can commit much below the Δ bound despite larger payloads.

Delay and System size. In this set of experiments, we measure the impact of the maximum network delay Δ on the performance, i.e., the response rate and commit latency of Sync HotStuff and Apollo.

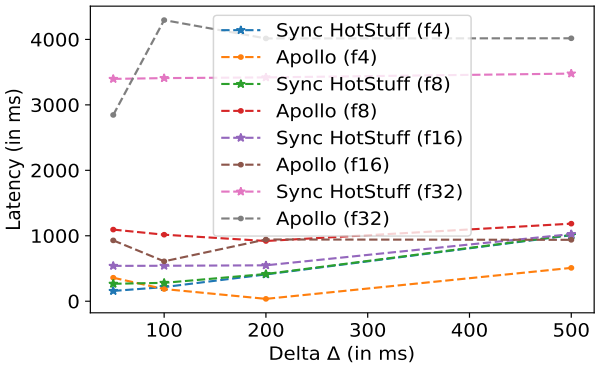
Remark. Sync HotStuff and Apollo behave differently for different values of w. In the original paper [3], the numbers reported have a hidden parameter w that varies, to obtain the points where the response rate is maximum and the latency is the minimum. With that strategy, Apollo gets the maximum throughput starting from very low w values, with the lowest latencies, but Sync HotStuff suffers for those values of w. Therefore, in this work, to be fair to both the protocols, we compare Apollo and Sync HotStuff for the same values of w, thereby observing different curves from what is presented in the original paper [3].

Experiment. We use the same setup as described in the previous experiments. We setup $n = 2f + 1$ instances for different values of faults f. For instance, in Figure 8a, a f32 curve uses $n = 65$.

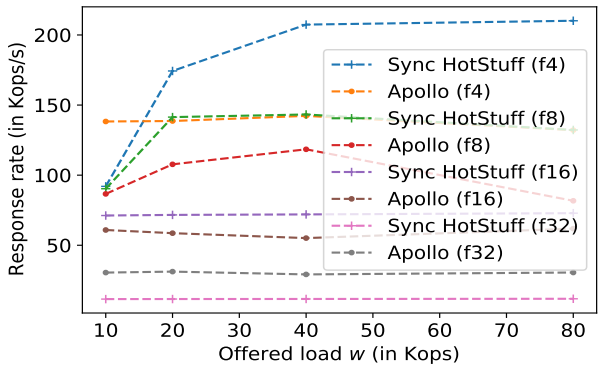
Figure 8a shows how the response rates for Sync HotStuff and Apollo react to changing worst case network assumption Δ. As expected, the response rate for Apollo is independent of the worst case network delay assumption. On the other hand, we can clearly observe Sync HotStuff's performance dropping as n increases. This is because at larger Δ parameters, Sync HotStuff needs



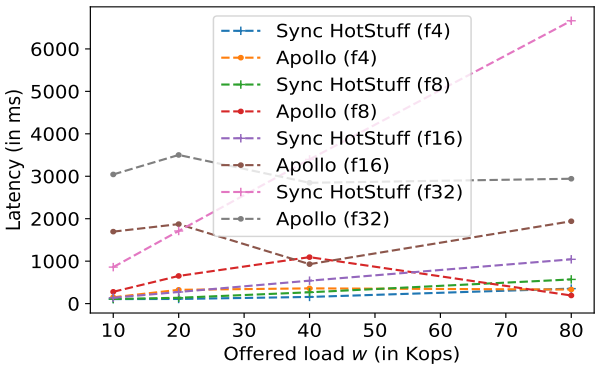
(a) **The network delay** (Δ) **vs. Response rate (Kops/s)** for various system sizes n. w is set to $10,000$.



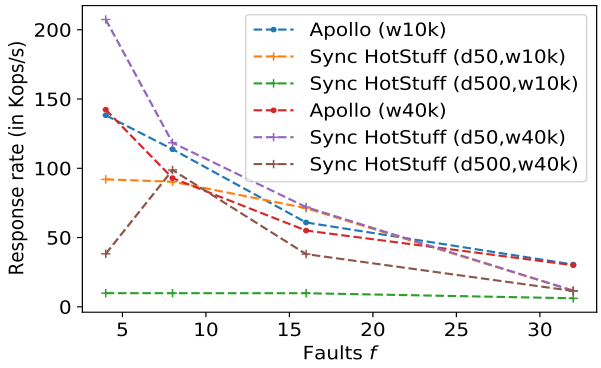
(b) **The network delay** (Δ) **vs. Latency**, for various system sizes n. w is set to $10,000$.



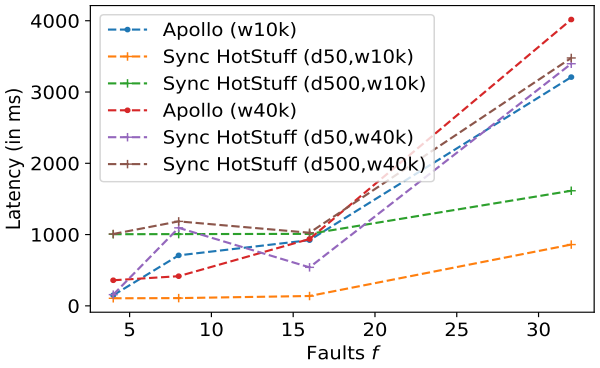
(c) **Offered load** (w) **vs. Response rate (in Kops/s)**, for various system sizes. Δ is set to 50 ms.



(d) **Offered load** (w) **vs. Latency**, for various system sizes. Δ is set to 50 ms.



(e) **Faults tolerated** (f) **vs. Response rate (in Kops/s)**, for various Δ and w parameters.



(f) **Faults tolerated** (f) **vs. Latency**, for various Δ parameters.

Figure 8: **Delay and System size statistics for Apollo and Sync HotStuff** with system parameters: faults $f = n/2$, block size $b = 400$ and payload $p = 0$.

larger w to prevent starving at thus remain at the optimal response rates. Also, note that for larger f such as 32, the response rates of Apollo is 3x more than that of Sync HotStuff. This is

because no matter the number of nodes, every block produces implies some block being committed, and therefore the response rate depends on how fast new blocks can be produced, whereas in Sync HotStuff a new block requires more and more signatures to form certificates which block the next proposal.

Figure 8b shows how the latency is affected by the worst case network delay parameter Δ. We observe that the general trend is for the latency to increase as n increases for both the protocols. For Sync HotStuff, the change is slow (due to the stable leader), whereas the impact of increasing f and therefore n is directly felt by the latency in Apollo. For instance, one can clearly observe the doubling of latencies in Figure 8b as f doubles from 16 to 32.

System load. In this experiment, we measure the performance of Sync HotStuff and Apollo with increasing system load.

Experiment. We use the same setup as described previously. This time, we measure the performance impact on performance (response rate and latency) with increasing the offered load (w).

Figure 8c demonstrates the performance of Sync HotStuff and Apollo with varying system load as n increases. From our experiments, we observe that Sync HotStuff is less tolerant to increases loads as n increases as the slope of the lines increase. Apollo however always manages to saturate the response rate and the block production.

Figure 8d demonstrates the performance with respect to latency. Apollo even though trades latency with increasing n performs better with increasing payloads than Sync HotStuff at higher system loads since all the proposals need to managed by a single leader. We also observe transactions occasionally being committed extremely fast (not shown in the figures) as sometimes the round robin executes perfectly leading to all proposers being ready to propose immediately.

Scalability. In this set of experiments, we measure the scalability of Apollo when compared to Sync HotStuff as f (and thus n) grows.

Figure 8e demonstrates the scalability of the system, to produce responses and f and thus n grows. We show these numbers for 2 different values of w, to show variation with w, and also for two different Δ parameters for Sync HotStuff. As f grows, we clearly observe a loss in response rate for all the protocols, at a fixed w. For Sync HotStuff we observe that for the same block size and same load, the throughput keeps falling. This is also true for Apollo, but its decline is more gradual.

Figure 8f presents the other half of the picture presented in Figure 8e. From Figure 7a, we know that for Sync HotStuff, eventually after some point the latency shoots up with no improvement in throughput indicating system saturation. Firstly, we observe these saturations at lower values of w as n increases. Therefore, looking back at Figure 8e, we can conclude that the throughput of the system indeed drops as n increases.

The Apollo protocol is no exception to this, however, we observe that the drop in response rates is not as drastic as that of Sync HotStuff, and can therefore sustain higher response rates as n increases, thereby indicating improved performance with increasing n, and thus scalability. We note here that this offers a trade-off. Depending on the use-case, if the system needs to sustain high response rates then Apollo is an ideal choice, however if latency is a concern then a deeper study may be required depending on the expected loads w.

7 CPS Performance

SMR also finds use in the distributed Cyber-Physical Systems (CPS). Distributed CPS are a connected network of embedded devices deployed in physical spaces to collect and process information.

These devices are light-weight in nature and run with limited resources. One of the major concerns in this setting is minimizing devices' energy consumption to last longer. An example of their usage is in the field of precision agriculture, where distributed CPS are deployed to monitor humanly unmanageable, large farmlands (the "sensing" aspect). These systems are also programmed to take affirmative actions in the form of adaptive actuation of sensors in response to various environmental conditions changes, making them prone to cyber-attacks. According to the recent U.S. Department for Homeland Security (DHS) report [7], intentional/malicious falsification of data can disrupt crop and livestock sectors. Along with this, the introduction of rogue data into a sensor network also has detrimental consequences like damaging crops or herds. SMR ensures the system's safety and liveness under Byzantine faults and is a crucial tool for building fault-tolerant CPS. However, existing SMR protocols [3, 28, 20, 12, 14] are not optimized for energy efficiency and are expensive to run on these devices.

Table 3: **Energy statistics for Apollo and Sync HotStuff CPS nodes** in the optimistic case

Protocol	**System size**	**Type**	**Energy (in mJ)**
Apollo	3	Leader	628.38
		Replica	886.2
	5	Leader	628.38
		Replica	886.2
Sync HotStuff	3	Leader	2954.03
		Replica	2972.53
	5	Leader	4796.88
		Replica	5103.15

We evaluated Sync HotStuff and Apollo in a distributed CPS environment. The environment consists of a distributed test-bed of $n = 3$, and $n = 5$ NUCLEO F401-RE embedded devices, communicating using Bluetooth Low Energy (BLE) advertisements as broadcast links. We used BLE advertisements to communicate between devices because they are energy-efficient communication media. Although BLE advertisements' are not a reliable way of transferring data, we use redundant transmissions to ensure high message delivery reliability. As the number of devices receiving the advertisements increases, BLE transmissions' redundancy factor will also increase, increasing the energy consumption. However, in our case, the number of devices does not vary by a considerable number, making it safe to assume that all the devices within the range of BLE communication to the leader will reliably receive messages without incurring additional energy expense. The devices use SHA256 for generating block hashes and sign messages using the Secp256r1 ECDSA signature scheme. We set the block size $b = 1$ with a transaction consisting of 16 bytes.

We implemented SMR on the test-bed using both Apollo and Sync HotStuff and measured the energy consumed per block using the `SALEAE Logic Pro 8` and the `INA-169` current sensor. We measured the energy consumed for the SMR by subtracting the average energy consumed by the device's OS in a sleep mode, for the same time. We report the energy measurements in Table 3.

From Table 3, we can clearly observe that Apollo consumes significantly lesser energy than Sync HotStuff, making it a more energy-efficient choice. The low energy consumption of Apollo is due to its constant per node signature and linear communication complexity. The energy consumption for Apollo does not change with respect to n because we are using BLE advertisement packets as

multicast links. The rate of increase of energy consumption with respect to n is higher for Sync HotStuff because of its quadratic signature and communication complexity per block commit.

8 Conclusion

In this work, we develop UCC Rule, that is efficient (energy-wise, communication and computation-wise) and does not rely on equivocation unlike classical synchronous protocols. We also develop Apollo, an SMR protocol based on this rule, that commits a block every δ, with a block latency of $O(f\delta)$ per block, uses relative timers and is view-change free in both the standard synchrony and weak synchrony models. We also show how to synchronize the head of the chain at any instant, using any Byzantine Agreement protocol.

Acknowledgements

We would like to thank Ted Yin for insightful comments and details on the internals and optimizations in the public implementation of HotStuff [22].

This work has been partially supported by the Army Research Laboratory (ARL) under grant W911NF-20-2-0026, the National Institute of Food and Agriculture (NIFA) under grant 2021-67021-34251, and the National Science Foundation (NSF) under grant CNS-1846316.

References

[1] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous byzantine agreement with expected $o(1)$ rounds, expected $o(n^2)$ communication, and optimal resilience. *Financial Cryptography and Data Security (FC)*, 2019.

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

[3] 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.

[4] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the libra blockchain. *The Libra Assn., Tech. Rep*, 2019.

[5] Piotr Berman, Juan A Garay, and Kenneth J Perry. Towards optimal distributed consensus. In *30th Annual Symposium on Foundations of Computer Science*, pages 410–415. Publ by IEEE, 1989.

[6] Adithya Bhat. adithyabhatkajake/libchatter-rs. `https://github.com/adithyabhatkajake/libchatter-rs/`.

[7] Aida Boghossian, S Linsky, A Brown, P Mutschler, B Ulicny, and L Barrett. Threats to precision agriculture. *Dept. Homeland Security (DHS)*, 2018.

[8] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.

[9] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on bft consensus. *arXiv preprint arXiv:1807.04938*, 2018.

[10] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[11] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[12] 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.

[13] T-H Hubert Chan, Rafael Pass, and Elaine Shi. Pala: A simple partially synchronous blockchain. *IACR Cryptol. ePrint Arch.*, 2018:981, 2018.

[14] T-H Hubert Chan, Rafael Pass, and Elaine Shi. Pili: An extremely simple synchronous blockchain. *IACR Cryptol. ePrint Arch.*, 2018:980, 2018.

[15] Determinant. Determinant/salticidae. `https://github.com/Determinant/salticidae`.

[16] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

[17] Sisi Duan, Michael K Reiter, and Haibin Zhang. Beat: Asynchronous bft made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2028–2041, 2018.

[18] Paul Feldman and Silvio Micali. Optimal algorithms for byzantine agreement. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 148–161, 1988.

[19] 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.

[20] Yue Guo, Rafael Pass, and Elaine Shi. Synchronous, with a chance of partition tolerance. In *Annual International Cryptology Conference*, pages 499–529. Springer, 2019.

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

[22] Hot-Stuff. hot-stuff/libhotstuff. `https://github.com/hot-stuff/libhotstuff`.

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

[24] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42, 2016.

[25] Nelson Minar. A survey of the ntp network, 1999.

[26] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

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

[28] 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.

[29] Build reliable network applications without compromising speed. `https://tokio.rs/`.

[30] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[31] 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 Weak Synchrony

Guo et al. [20] first proposed the χ-weak synchrony model, and PiLi [14] first proposed an SMR protocol in the weak synchrony model where rounds proceed in units of Δ. Sync HotStuff [3] renamed the model to *mobile sluggish fault* model while adapting χ-weak synchrony (informally) to the non-lock step model, and deals with corruptions at instants of time, instead of rounds as in PiLi [14].

Liveness Assumptions. Finally, in the weak synchrony model, an assumption of persistently online nodes is necessary to guarantee any form of eventual progress, i.e., liveness. This is analogous to the Partial Synchrony - Type II liveness parameter called GST (Global Stabilization Time). The weak synchrony model also assumes two relaxed GST assumptions in order to guarantee liveness. The GST GST used in partial synchrony, assumes the existence of some time GST GST unknown to the system, after which the network holds the standard synchrony assumption for all correct nodes.

The *strong-GST* parameter in the weak synchrony model, defined in Definition A.1, models a similar liveness assumption, and assumes that all the correct nodes become persistently online after some time GST_s unknown to the nodes beforehand (and cannot be estimated).

Definition A.1 (Strong - Global Stabilization Time [20])**.** *Strong - Global Stabilization Time* GST_s*, is defined to be the first round after which all correct nodes are online, i.e., communicate with each other in* Δ *time.*

The strong-GST parameter can be a very strong restriction to impose on the adversary. In order to weaken this restriction, PiLi [14] introduces a weaker parameter, which we term the *weak-GST* parameter. This parameter, defined in Definition A.2 dictates that there must exist some fixed $n - f$ nodes, which are persistently online after time GST_w. The adversary is still allowed to place other extra correct nodes in the partition.

Definition A.2 (Weak - Global Stabilization Time [20])**.** *Weak - Global Stabilization Time* GST_w*, is defined to be the first round after which there exist more than* $n/2$ *correct nodes (denoted by* $\mathcal{O}$*), which are persistently online afterwards.*

B Implementation Details

We implement protocols in Rust due to its strong memory safety and program correctness guarantees. We use the tokio runtime to drive our protocol. Our codebase is inspired from the design in Diem [4] as well as Sync HotStuff. We implemented two variants of Sync HotStuff: a stable leader and a round robin variant, and Apollo in this framework for a fair comparison. Analogous to the implementation in Sync HotStuff, we also treat a certificate as a compact vector of `ED25519` signatures. Our entire codebase consists of $7.3K$ lines of Rust code. The consensus modules consists of about ≈ 800 lines of Rust code for Apollo, and ≈ 1.3K lines for Sync HotStuff, both of which includes consensus code for the clients as well as the nodes.

We choose to re-implement Sync HotStuff (stable leader and round-robin) instead of using the authors' public implementation, for the following reasons:

1. The public implementation is adapted from HotStuff [31], a partially synchronous SMR protocol, yielding a more complex implementation than necessary. We rewrote Sync HotStuff from scratch to build a cleaner implementation.

2. The public implementation is optimized for stable leaders, and updates the transaction pool only on commit. Employing this policy, for Apollo is not optimal, as nodes will update the transaction pool after $f + 1$ blocks are proposed with the same transactions. Changing this policy, in the public implementation also requires a major re-write of their implementation.

3. The underlying networking library `salticidae` [15] only allows multicasts and does not provide an interface to send point-to-point messages, which we require for Apollo. Therefore, we have to re-write the underlying networking library as well.

4. The client for Apollo is special since it can also apply the *UCC* rule on its own and decide finality on its own. However, the client in Sync HotStuff and HotStuff uses $f + 1$ acknowledgements over TLS connections as a mechanism to ensure finality of the blocks.

5. We want, and have created, a general purpose synchronous networking library with a networking module that is protocol agnostic, a consensus layer that is network agnostic, and a crypto module that is instantiation agnostic, i.e., does not know what the underlying digital signature scheme is. With the compile time guarantees from Rust, our codebase is also guaranteed to be thread safe and memory-leak free.

6. Our Rust codebase is of independent interest for implementing other synchronous cryptographic/distributed system protocols.

Optimizations. We perform several optimizations in our implementation so that we get the maximum performance from both implementations. In particular, we perform the following optimizations:

- *Runtimes.* We use the tokio runtime [29] for our implementation. This allows us to spawn as many non-blocking IO tasks as we want, and the runtime will schedule available tasks on CPUs. Tokio is also efficient for networking as it lets the operating system wake the process up when sockets are ready for use (read and write).

- *Tasks.* With Tokio for both Sync HotStuff and Apollo, we spawn a task that is responsible for communication with a peer (a client or a protocol node). We add manager tasks that read messages from any peer that has a message, and writes to peers whenever a message is ready to be written. With tokio, multiple messages can be written to/read from sockets in parallel.

- *Reactors.* We implemented separate reactor/runtimes for the protocol networking module, the client networking module, and the consensus module. This prevents starvation, as we have clients that are constantly bombarding the server with transactions, and this must not slow down the consensus protocol or receiving messages from the other protocol nodes.

- *Channels.* We observed that the implementation of a channel mattered to the protocol. For example, Sync HotStuff performed better with a particular channel implementation (tokio channels), whereas Apollo performed better using futures channels. We used the channel implementation that is most favorable to the protocol.

- *Non-blocking proposal.* Our Sync HotStuff and Apollo implementations carefully spawn tasks, so that a proposal is not blocked by long tasks. For instance, when including payloads for transactions, we ship a committed transaction to a separate channel that adds payload and communicates with all the connected peers. This way, the propose step finishes quickly and does not need to wait until the committed block with the payload is received by the client.

- *Porting optimizations.* We also implemented and ported all the optimizations found in the existing public implementation of Sync HotStuff. For instance, we used a request-response paradigm, where only hashes are used everywhere except during a block proposal. If the block is already present, this saves significant network I/O as the node can query its local chain to get the block. The voting step in Sync HotStuff and the relay step in Apollo uses this optimization.

- *Lock-free.* Our code design avoids using locks such as Mutex, Semaphore or Read-Write Locks, but instead uses message passing and is designed to ensure that no two threads need to share data. However, for big pieces of data such as proposals, blocks, and config files we use atomic reference counted objects, which gives all threads a shared immutable reference without having to copy the underlying object.

- *Reduce network I/O.* We reduce the number of fields sent over the network. For instance, in a block, we have to compute the hash to check the validity of the block, and so we do not send the hash of a block over the network, but we fill it on getting a new block.