The Unique Chain Rule and its Applications*

Adithya Bhat[1], Akhil Bandarupalli[1], Saurabh Bagchi[1], Aniket Kate[1,2], and Michael K. Reiter[3]

[1] Purdue University, West Lafayette IN 47906, USA
{abhatk,abandaru,sbagchi,aniket}@purdue.edu
[2] Supra
[3] Duke University, Durham NC 27708, USA
michael.reiter@duke.edu

Abstract. Most existing Byzantine fault-tolerant State Machine Replication (SMR) protocols rely explicitly on either equivocation detection or quorum certificate formations to ensure protocol safety. These mechanisms inherently require $O(n^2)$ communication overhead among n participating servers. This work proposes the Unique Chain Rule (UCR), a simple rule for hash chains where extending a block by including its hash in the next block, is treated as a vote for the proposed block *and its ancestors*. When a block obtains a vote from at least one correct server, we can commit the block and its ancestors. While this idea was used implicitly earlier in conjunction with equivocation detection or quorum certificate generation, this work employs it explicitly to show safety.

We present three applications of UCR. We design *Apollo*, and *Artemis*: two novel synchronous SMR protocols with linear best-case communication complexity using round-robin, and stable leaders, respectively as the first two applications. Next, we employ UCR in a black-box fashion toward making any SMR commits publicly verifiable, where clients will no longer have to wait for $2f + 1$ confirmations on every block, where κ is a security parameter and f is the number of Byzantine faults tolerated by the protocol, but can instead collect a UCR proof consisting of $\min(\kappa, f) + 1$ extensions on a block. This results in faster syncing times for clients as the publicly verifiable proofs can also be gossiped with every new block extension confirming a new block.

1 Introduction

State Machine Replication (SMR) [35] is a fundamental distributed-computing primitive that is receiving renewed attention due to its potential to support blockchains. At its core, an SMR protocol coordinates a set of n servers running a deterministic service so that they collectively implement the abstraction of a single, correct server, even when a subset of servers turns malicious (or Byzantine). Most SMR protocols [1–4,8,12,14–16,18,24,27,28,31,36–38,42] achieve this coordination of forming a sequence/chain of blocks (of instructions/transactions)

* An extended version is available at https://eprint.iacr.org/2021/180

using a *leader* server that the other servers follow, with provisions to change this leader in response to some faults or regularly by design.

In the standard (bounded) synchronous communication setting with the worst-case network delay of Δ for messages, publicly-verifiable Byzantine fault-tolerant (BFT) SMR protocols can tolerate up to one-half Byzantine faults.[4] Many synchronous SMR protocols [2, 3, 14, 16, 24, 36] achieve this resilience primarily using the lack of equivocation in $O(\Delta)$ time; here, confirming lack of equivocation for a message (or a block) requires sending the message to all the servers and then not hearing any complaints in 2Δ time. Other synchronous protocols [16, 36] that avoid the above equivocation detection use the fact that at most one message can obtain $3n/4$ votes. Nevertheless, they still require certificates with $O(n)$ signatures, and incur quadratic in n communication.

This work explores a significantly different approach towards SMR, which is reminiscent of proof-of-work SMR systems [33, 40] such as Bitcoin. The Bitcoin networks follow an informal rule that after observing six blocks of transactions 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 server is considered to be small enough. We observe that if we can ensure that no alternate chain of blocks is possible in the permissioned SMR systems, i.e., the SMR chain we have is unique, then we can use the *unique chain* to commit blocks. Subsequently, we ask the following question: *How many blocks do we need to observe before we are sure that a block B is final, in a permissioned network?* The answer turns out to be γ for a protocol if: (i) the γ blocks contain blocks from at least one correct server, (ii) there is only one server that proposes a block for a height, (iii) correct block proposals are always accepted, and (iv) we use a tamper-resistant chain (e.g., hash-chain, where every block contains the hash of the previous block). Based on these observations, we develop a consensus rule called the *Unique Chain Rule (UCR)* (Section 3) and its three applications: two novel SMR protocols: Apollo (Section 4) and Artemis (Section 5), and a protocol to make any SMR publicly verifiable (Section 6).

At network speed with delay δ, our protocols commit a block every δ time, with a constant[5] per-block commit latency of $(\min(\kappa, f)+1)\delta$, and rely on Δ only to detect crashed leader(s). Our protocols are the first synchronous SMR protocols with certificate-free optimistically linear communication when the leader(s) behave correctly. It also produces $2\times$ more blocks as the time between two successive produced blocks, i.e., block period is $1/2$ of the state-of-the-art protocols due to the lack of a round-trip communication to form quorum certificates. Our protocols are efficient in terms of cryptography (see Table 1), making it a suitable candidate for SMR in resource-restricted environments.

[4] It is possible for SMR protocols to tolerate more than $1/2$ faults. However, these SMR protocols cannot safely convince any external observer of statements regarding the latest state of the system due to the dishonest majority [32].

[5] Many related works claim constant latency [1, 3]. The correct term should be $(\min(\kappa, f)+1)$ as leader randomization is inherently assumed and for small f round-robin protocols are sufficient.

Table 1: **Comparison of the best case (i.e., all the servers are correct) and worst case of Apollo with the related synchronous SMR works. Here $\hat{\kappa}$ is $\min(f, \kappa)$.**

Protocol	Best Case				Worst Case		
	Commit Latency	#Sign	CC	Block Period	Latency	#Sign	CC
Dfinity [2, 25]	$6\Delta + 2\delta$	$O(n)$	$O(n^2)$	2Δ	$O^\star(\hat{\kappa}\Delta)$	$O(\hat{\kappa}n^2)$	$O(\hat{\kappa}n^3)$
PiLi [16]	26δ	$O(n)$	$O(n^2)$	2δ	$O(\hat{\kappa}\Delta)$	$O(\hat{\kappa}n)$	$O(\hat{\kappa}n^2)$
Sync HS [3]	6δ	$O(n)$	$O(n^2)$	2δ	$O(p^\star\Delta) + O(\hat{\kappa}\Delta)$	$O(\hat{\kappa}n)$	$O(\hat{\kappa}n^2)$
Rot. SMR [5]	$2\Delta + 2\delta$	$O(n)$	$O(n^2)$	2δ	$O(\hat{\kappa}\Delta)$	$O(\hat{\kappa}n)$	$O(\hat{\kappa}n^2)$
Streamlet [14]	$8\Delta + 8\delta$	$O(n)$	$O(n^2)$	2Δ	$O(\hat{\kappa}\Delta)$	$O(\hat{\kappa}n)$	$O(\hat{\kappa}n^2)$
$1-\Delta$ SMR [4]	$1\Delta + 2\delta$	$O(n)$	$O(n^2)$	2δ	$O(p^\star\Delta) + O(\hat{\kappa}\Delta)$	$O(\hat{\kappa}n)$	$O(\hat{\kappa}n^2)$
OptSync [36]	2δ	$O(n)$	$O(n^2)$	2δ	$O(p^\star\Delta) + O(\hat{\kappa}\Delta)$	$O(\hat{\kappa}n)$	$O(\hat{\kappa}n^2)$
Apollo	$(\hat{\kappa}+1)\delta$	$O(1)$	$O(n)$	δ	$O(\hat{\kappa}\Delta)$	$O(\hat{\kappa}n)$	$O(\hat{\kappa}n^2)$
Artemis	$(\hat{\kappa}+2)\delta$	$O(1)$	$O(n)$	0	$O(\hat{\kappa}\Delta)$	$O(\hat{\kappa}n)$	$O(\hat{\kappa}n^2)$

 # Sign is the number of signature generated by all the servers per proposal/block. The number of verification operations for each protocol is n times the signing complexity as every signed message is verified by all the servers. **CC** stands for Communication Complexity of the protocol. **Block Period** is defined as the time between two successive block proposals. $O^\star(g)$ denotes $O(g)$ with high probability. $p^\star$ denotes the number blocks proposed before the leader crashes. In Sync-HotStuff [3] and OptSync [36], a leader is blamed only if p blocks are not proposed in $(2p+4)\Delta$ time. If p' blocks are proposed by time t, then the servers wait for $p^\star = (2p'+4)\Delta - t$ time before blaming the leader.

UCR can be applied to make any SMR commit *publicly verifiable*, i.e., any server that observes the SMR commit data can non-interactively confirm the correctness of the commit. Such publicly verifiable commits can be leveraged to efficiently disseminate the state and prove the state to the clients instead of requesting $2f+1$ acknowledgments (of which at least $f+1$ are guaranteed to be from correct servers) from the servers for every block.

1.1 An Informal Exposition of Key ideas

Unique Chain Rule (UCR). As an example, consider a system of n servers with up to f Byzantine servers, where the blocks proposed in every round are from a round-robin among all the servers. Assume that the blocks use a hash-chain, i.e., a block B proposed in round r includes the hash of its parent block from the previous round $r-1$. Implicitly, this proposer is voting for all the blocks in rounds $\{r, \ldots, 0\}$. In contrast to existing SMR protocols where quorum certificates (i.e., a vector of signatures from more than 50% or 66.67% of the servers) were built for every block of every round, we can use these implicit votes to form certificates for blocks. A traditional certificate guaranteed that no other block for the same height can get certified, while our implicit certificates guarantee that no other chain with the same prefix can form, making the prefix a *unique chain*. We present the resulting commit rule as the Unique Chain Rule (UCR).

Apollo Protocol. Using UCR, we then develop an SMR protocol. We use random leader selection to ensure that at least one leader is correct in any sequence of $\kappa + 1$ rounds. We add a constraint that a server must extend a block from the previous round unless it can obtain a certificate consisting of $n/2 + 1$ signatures claiming Byzantine behavior. We use this to ensure that a block proposed by a correct server cannot be skipped by Byzantine servers. Now, if any server (including the client) observes a chain that is $\kappa + 1$ long, it knows that one of those servers is correct, and its block will never be skipped. Therefore, the chain is unique and final; thus, it can be committed.

In the optimistic conditions, i.e., when the leader(s) is correct, Apollo protocol creates new blocks to increase the length of the chain and thus commit blocks without equivocation detection. In this setting, it is sufficient for all the servers to forward the latest block to the next leader. This gives us certificate-free optimistic linearity and allows responsive commits, i.e., speeds independent of Δ.

Round-robin protocols are efficient in distributing the system load across all the participating servers and are also used to ensure chain quality, i.e., the majority of the chain is from the correct servers. However, Byzantine servers can slow the progress of the SMR by crashing and slowing down the pipeline.

Artemis Protocol. In order to overcome this, we present a stable leader-based SMR protocol: Artemis protocol. In a nutshell, in Artemis protocol, there is a dedicated leader server that creates blocks. The other servers run a modified version of Apollo using the latest leader's blocks. If the leader crashes, the protocol changes the *view* and elects a new leader. Since the latest block is used, a slow proposer in the inner Apollo protocol may stall for some time, but when the next correct server proposes, it will propose the latest block thus effectively catching up with all the servers to the highest block.

Publicly Verifiable SMR. In several permissioned widely-deployed SMR implementations, committed blocks or states are downloaded by clients by connecting to the servers and waiting for $f + 1$ acknowledgments for the block [6, 17, 34, 39]. This incurs a significant overhead on the servers for large numbers of clients. If the chain is ℓ blocks long, the cost incurred by the servers is $O(\ell f)$.

A typical approach to solving this is to add another step of quorum certificate generation after committing in every round, and gossip this quorum certificate to all the clients. This approach incurs $O(1)$ signature generation overheads and $O(f)$ signature verification overheads for all the servers in the system. It also incurs $O(1)$, and $O(f)$ certificate verification overheads respectively, with and without the usage of threshold signatures, for the clients.

Using UCR, we can make the servers gossip a signed message after committing, in every round. On collecting *any* increasing sequence of $\kappa + 1$ such signed state messages, any client (without talking to the servers) can verify that the state is correct leading to $O(1)$ signature generation overhead, $O(1)$ signature verification overheads, for all the servers in the system, and $O(\kappa)$ signature verification overheads for the clients, irrespective of the usage of threshold signatures.

This application provides a trade-off to the publicly verifiable SMR problem with fewer overheads on the servers and more overheads to the clients.

1.2 Related Work

Recently, several permissioned SMR protocols have emerged, in the standard synchrony [3, 5, 14, 16, 36], weak synchrony [3, 24], partial synchrony [10, 15, 20, 21, 37, 38, 43], and asynchronous models [18, 27, 28, 31][6]. Permissionless systems such as Proof-of-Stake (PoS) blockchain protocols require a rotating leader based SMR, where the leader is generally chosen randomly with probability of being a leader for an epoch/round being directly proportional to the amount of stake invested. Therefore, permissioned consensus protocols are of interest in this area. We discuss the landscape of Proof-of-Work and Proof-of-Stake protocols.

PoW and PoS. In retrospect to this work, UCR can be viewed as being implicitly applied in Proof-of-Work [33,40] and Proof-of-Stake [10,19] based systems, which use the fact that votes on hash-chain or checkpoints in the directed acyclic hash graph of blocks also serve as votes for prior checkpoints or blocks. In particular, Casper [10] uses the fact that if a validator vote for two conflicting checkpoints then its stake is slashed. Here, the conflicting checkpoint is implicitly determined by checking two votes that differ in their ancestors.

In the next part of our literature review, we focus on works that are similar to our work and use standard synchrony assumptions.

BFT-SMR Protocols. The applications of UCR in the literature have always been in secondary roles as a helper mechanism to equivocation or quorum certificate based commit rules [1–3, 10, 18, 20, 27, 28, 31, 33, 36, 37]. For instance, the idea of using a vote on a block in a hash-chain as votes for all its parents has been used implicitly in [2,3,16,36]. In Sync-Hotstuff [3], Abraham et al. mention that for a hash-chain *"the voting step on a block also serves as a voting step for all its ancestor blocks that have not been committed"*. While several protocols [1–3, 10, 18, 20, 27, 28, 31, 36, 37] use this for committing ancestors of a committed block, none of them build an explicit protocol out of this observation. They use a UCR-like idea whereby adding extra markers to vote messages for a block B, the vote messages are used as endorsements (a vote) for that block and its ancestors, and when an ancestor gets x endorsements it becomes x-strong. We dive deeper into a few protocols to illustrate this.

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* to refer to weak synchrony. All of their protocols use a fixed leader and run 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 round-trip gives rise to the extra δ between two successive block proposals. Since all of these steps occur in parallel, two consecutive proposals are only delayed by 2δ.

[6] This list is not exhaustive.

OptSync. OptSync [36] is very similar to Sync-HotStuff, except servers can commit synchronous or optimistically, simultaneously. If 2Δ passes after multicasting a block, the commit is called a synchronous commit. If a block receives $3n/4$ votes, then the servers commit the block, and this mode of commit is called a responsive commit. They also present a responsive view-change protocol.

In contrast to both Sync-HotStuff and OptSync, Apollo and Artemis output two and several, respectively, blocks for every block proposed, with a net $2\times$ theoretical improvement in the throughput. Comparing block latencies with OptSync, our protocols are slightly worse in the constants. However, our protocols are linear in optimistic conditions. We treat OptSync as our baseline since it has the lowest latency and block period.

PiLi. PiLi [16] proposes a blockchain (SMR) protocol. 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. [16], if the leader of epoch $r+1$ cannot get a strongly notarized block (greater than $3n/4$ votes) but only a notarized block (greater than $n/2$ 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. 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.

1Δ-SMR. 1Δ-SMR [4] uses the term *good-case latency* to refer to optimistic conditions when the leader is correct and the messages are delivered instantly, i.e., $\delta \approx 0$. It explores the lowest commit latency achievable and shows a lower bound of 1Δ. It also presents an SMR protocol that has almost-optimal $(1\Delta + 2\delta)$ commit latency. However, it does not support optimistic responsiveness. They show a Byzantine Broadcast protocol with the good-case latency of $1\Delta + 1\delta$ if all the servers start at the same time (called as *synchronized* start), and $1\Delta + 1.5\delta$ otherwise. They use a black-box BA protocol at the end to finish the protocol, and therefore, it is not clear how to use it as an SMR protocol efficiently, besides running the Byzantine Broadcast repeatedly. All the above synchronous 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.

Optimistically responsive synchronous protocols [3, 16, 36] support a mode called *optimistic responsiveness*. In this mode, they assume $> \frac{3}{4}n$ servers along with the leader(s) are correct and the network delivers messages for the correct servers in $O(\delta)$, which allows the protocols to commit in $O(\delta)$.

Note on Optimistic Responsiveness Assumptions. The requirement for optimistic responsiveness is from Sync-HotStuff [3] and OptSync [36]. They require the condition that the leader and any $3n/4$ servers are correct. Our require-

ment for constant latency is that the view leader (if any) and the proposer set (consisting of $f+1$ servers) is correct. Our requirement is incomparable (slightly stronger) to the requirement from Sync-HotStuff and OptSync. PiLi [16] has the strongest requirement that all the leaders (and thus all the servers) of an epoch are correct for optimistic responsiveness.

We focus on another related work whose protocol flow is similar to our protocol in the communication pattern but have some significant differences. *BChain.* BChain [20] is a partially synchronous SMR protocol where the servers are arranged in a chain and then the block is passed from the head to tail, and an acknowledgment message in the reverse direction. This incurs $O(n)$ communication over $O(n)$ rounds to commit 1 block, and multiple blocks can be processed in parallel under optimistic conditions. They use re-chaining and view-change techniques if any server suspects any server of slowing the chain. With respect to this, our protocols have constant latency and more efficient pipelining, leader rotation, and view-change.

Our Apollo protocol produces two times the number of blocks produced by the state-of-the-art synchronous SMR protocol OptSync [36] while having only $O(1)$ signature complexity. Our second Artemis protocol can produce and commit as many blocks as the leader can sign, thus leading to a block period of 0. Unlike threshold signature-based protocols [1, 2, 23, 36, 43] our protocols can achieve linear communication complexity without this assumption. This allows our protocols to be used in the post-quantum settings where no threshold signature schemes are currently known.

Sanity Check with Bounds. Finally, we perform a sanity-check of our results with the existing literature about lower and upper bounds in SMR. Shrestha et al. [36, Theorem 1] show that if the optimistic commit latency is x when tolerating more than $n/3$ Byzantine faults, then when $n/2-1$ servers crash the commit latency must be $2\Delta - x$. We are subject to this bound, as in the non-optimistic conditions, our commit latency is $O(\Delta)$.

Abraham et al. [4] use the term *good-case latency* to denote the condition when a fixed leader and $< n/2$ servers are Byzantine and the network delivers messages in time $0 \approx \delta$ [4, Definition 1]. It is impossible to have a good-case latency below Δ [4, Theorem 3]. We are also subject to this bound, as there exist adversarial strategies where our commit latencies are $O(\Delta)$.

Strengthened-BFT. Strengthened-BFT [41] develop a protocol based on HotStuff. In this protocol, the client chooses the number of faults x that it believes to be corrupt in the system. The protocol guarantees that if a block is x-strong committed, then no other block can be x'-strong committed, where $x' \geq x$ in the presence of $t \leq x$ Byzantine faults. They achieve this by adding *endorsements* for every block B, which are either direct vote messages for B or indirect votes for B by votes on B' which extends B. Observe that this is similar to our approach, however, they use it to achieve a client-belief diversity [30], in partially synchronous settings, and using explicit vote messages for blocks, whereas we build a new commit rule and use the extension itself as a vote to its parents (thereby making the vote implicit). Their protocol gains linearity using threshold

signatures, and still uses certificates while we have linearity independent of the threshold signature assumption.

Algorand. Algorand [22] is also a sublinear $O(n \log n)$ SMR protocol that assumes strong synchrony, i.e., there exist periods (say 5% of the times) where the network is synchronous. They use the idea of *cryptographic sortition*, i.e., use VRFs to select a subcommittee (only the winners know that they won) of size $c = \log n$ and these servers run a Byzantine agreement protocol among themselves. The block period for this protocol is 2δ (1δ to declare to everyone that they were elected, and another 1δ to broadcast the finalized block) along with the amount of time required to finish the BA.

2 Preliminaries

Our system consists of a set $\mathcal{N} := \{p_1, \ldots, p_n\}$ of n servers with $f < n/2$ Byzantine servers with static corruptions[7]. A server is *correct* if it is never Byzantine.

Setup. We assume secure $(n, n/2+1)$-threshold digital signatures (e.g., BLS [7]) and denote signed messages from p_i by $\langle \cdot \rangle_{p_i}$, and the aggregated threshold signature on the same message m as a (quorum) certificate $\mathcal{C}(m)$ similar to most other SMR protocols (such as [3, 15, 16, 25, 36, 43]). We assume that all the servers use the same genesis block before starting the protocol which can be derived from a Common Reference String (CRS) setup. We also use the CRS to randomize our leaders as done by existing works [1–3].

We assume a fully connected standard (bounded) synchronous network which assumes a public worst-case network delay Δ, i.e., if a correct server sends a message to another correct server, then the message is received by the latter within Δ time from when it was sent by the former. Similar to most recent synchronous SMR protocols, we use two delays: Δ and δ. Δ refers to the synchrony bound, i.e., the *worst case network delay*, and δ refers to the optimistic (actual/real) network speed[8]. A *multicast* means a send-all operation where a server p_i sends a message to all servers $\mathcal{N}$.

State Machine Replication—SMR. An SMR protocol (Definition 1) executes transactions from clients using a state machine replicated across different servers. Clients are nodes that can be the servers themselves. The SMR protocol is typically implemented by generating a linearizable log of transactions. A secure SMR protocol guarantees two properties: *safety*, and *liveness*. Safety, in a broad sense, ensures that the states of the servers must be consistent, i.e., no two correct servers output different states at any point. Liveness, in a broad sense, argues that the system can never go into a deadlock.

[7] Our protocol is adaptively secure, but a different randomization protocol will be needed. There is a trade-off between constant latency and increased signature complexity using [11], or $O(f\delta)$ latency and constant signature complexity using round-robin.

[8] In practice, δ varies between pairs of servers, instances of time, and size of the message. However, the analysis here assumes that a single δ value is the optimistic delay time, a violation of which implies that we are not in the optimistic scenario.

Definition 1 (SMR [3]). *Assume a system of n servers $\mathcal{N} := \{p_1, \ldots, p_n\}$, f of which are Byzantine. The SMR protocol implements a linearizable log of transactions from clients with the following properties:*

1. ***Safety.*** *If two correct servers p_i, $p_j \in \mathcal{N}$ commit transaction tx and tx', respectively, at the same log height k, then $tx = tx'$.*

2. ***Liveness.*** *Each client transaction is eventually processed by the system.*

Chains and Blocks. The servers agree on a *chain* $\mathcal{C} := \{B_0, \ldots, B_\ell\}$, which we define as a list of blocks[9], where blocks contain client transactions. The height of a block is the index in this list or the chain. A block at height k is B_k. In particular, the first block B_0 is the genesis block with height 0. A block $B_k := \langle h_k, \mathsf{cmds} \rangle_L$ includes the hash of B_{k-1} as $h_k = H(B_{k-1})$ along with a list of transactions cmds. B_{k-1} and B_k share a parent-child relationship. h_k is the parent hash or pointer. Block $B_{k'}$ at height $k' < k$ is an ancestor of B_k as long as they have valid parent hashes linking them.

The genesis block is always *valid*. The child B_k of a valid block B_{k-1} is valid, if h_k is correct, and it satisfies other validity conditions imposed on cmds. A valid chain $\mathcal{C} := \{B_0, \ldots, B_\ell\}$ is a list of valid blocks starting with the genesis block B_0. The chain size is the highest height of blocks in the chain, i.e., $\ell = \mathsf{height}(\mathcal{C}[-1])$.
Tamper-resistance. Since the blocks in a chain are hash-linked, it is not possible to change a block in the chain without changing all the blocks after it. We call this the *tamper-resistance* property of the chain (Lemma 2).

3 Unique Chain Rule (UCR)

A quorum [29] is a subset of servers. In distributed protocols, we typically need a certain number of acknowledgements on a message to ensure that the other servers are in sync. We typically deal with $f + 1$ sized quorums in standard synchrony (e.g., [1, 2, 4, 36]) or $n - f$ quorums in non-synchronous[10] networks (e.g., [8, 10, 13–15, 43]). In these quorums, the names of servers are not as important, when compared to their count. A quorum certificate is a publicly verifiable message consists of these specified number of signatures from a quorum, typically instantiated with threshold signatures.

Synchronous SMR protocols [1, 3, 14, 16, 36] typically improve the fault tolerance from $n > 3f$ to $n > 2f$ by adding equivocation detection which involves $O(\Delta)$ waits due to the message delivery guarantees [14]. We observe that, in a hash chain, equivocation is a chain fork (multiple valid chains), and resolving equivocations translates into a fork-resolution problem. If we want to avoid equivocation detection, we need a mechanism to resolve chain forks.

For a system tolerating f Byzantine servers, $f + 1$ quorum certificate on a block is insufficient to remove equivocation detection of the block. A Byzantine proposer p_L can propose two blocks B and $B^\star$. If two correct servers vote for

[9] We use the notation from Python.

[10] Non-synchronous includes partial synchrony, asynchronous networks, etc. that are not standard synchrony.

B and $B^\star$ respectively, without being aware of the existence of the other block, then with the votes from the f Byzantine servers, both the blocks can obtain a quorum certificate.

Unique Chains. Let γ be a parameter such that in any sequence of γ rounds, there is at least one correct leader. Consider a protocol that uses round-robin leaders who propose one block in every round using hash chains. Trivially, this protocol has $\gamma \leftarrow f+1$. Consider that a server votes for a block by extending it in its turn to propose, instead of the traditional approach of voting for every block and building quorum certificates and detecting equivocation for them. Let *chain weight* of a block be the number of unique servers extending a block. Finally, if we can ensure that a correct proposer's block for a round is always extended by the correct servers, i.e., a Byzantine leader cannot propose a block without extending the block from the previous round if the previous leader was correct, then observe that when this chain weight exceeds γ for a block, no other valid chain can be formed that does not extend this block. Intuitively, if this was not true, then the Byzantine servers managed to overwrite a correct proposer's block thus leading to a contradiction.

We can ensure that a correct server's block for a round is always extended by the correct servers by changing the rejection condition: a valid block can be rejected only if there are $n/2 + 1$ explicit complaints against it. By explicit complaints, we mean $n/2 + 1$ signed (blame) messages for the round.

In the consensus literature so far, certificates consisted of signatures on a particular message/block, and used $O(1)$ such quorum certified blocks in the commit rules. The examples include $3n/4$ quorum with 1 certified block [3,16,36], $2n/3$ quorum with 3 blocks [13, 43], and $n/2 + 1$ quorum with 6, 13 blocks [14, 16]. However, we can look at the γ weighted chain suffix as equivalent to the $f + 1$ quorum certificate for the prefix of the chain, thereby leading to implicit certificates of size $O(\gamma) = O(\kappa)$. Using this certificate, we can ensure that the block, and thus the corresponding chain referenced by the block is *unique*, i.e., no alternate chain can form by the protocol. Definition 2 specify the requirements formally. In the rest of the paper, unless otherwise specified, we use $\gamma \leftarrow f + 1$.

Definition 2 (γ-UCR requirements). *The requirements to apply γ-UCR in a protocol: (1) the chains built are tamper-resistant, (2) blocks are proposed by servers such that there is at least one correct server in* any *sequence of γ rounds, and (3) a correct server's blocks are always accepted by all the correct servers.*

We state this formally in Theorem 1 as the Unique Chain Rule (UCR).

Theorem 1 (Unique Chain Rule). *Consider a protocol for n servers tolerating f Byzantine faults, and satisfying Definition 2. Then, on observing a valid chain $\mathcal{C} := \{B_0, \ldots, B_\ell\}$ of size ℓ (with $\ell > \gamma$), commit the prefix chain $\mathcal{C}[: \ell - \gamma]$.*

4 Apollo Protocol

In this section, we present the Apollo protocol which uses UCR (Theorem 1) to build a pipelined, linear SMR protocol for the standard synchrony model.

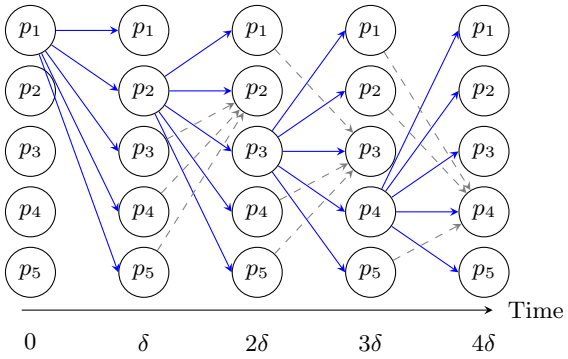

Fig. 1: **Overview of the Apollo Protocol in the optimistic case, when all the leaders are correct.** The blue messages are block proposals. The dotted lines are relay messages. The proposer for round $r+1$ can immediately propose as soon as it receives the block for round r. Hence, Apollo has a block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

Proposer Set. We define a proposer set $\mathcal{P}$ consisting of all (or $f+1$) servers $\mathcal{N}$. Let R be a random number chosen in the setup. We use a well-known technique [14, 16] and use $H(R, i)$ to randomly elect the leaders from $\mathcal{P}$ in every round. As servers agree on *misbehavior* from leaders (by committing blocks that contain proof of equivocation/no progress of leaders), we remove (or replace) the servers from the proposer set. This allows us to eventually stabilize on a set of leaders of size at least $f+1$ (even if $n \neq 2f+1$) that are correct.

4.1 Overview

We give an overview of Apollo in Fig. 1 and the technical details in Fig. 2. The protocol proceeds in rounds. p_1 is the leader for the first round and performs the *Propose* Step (Step 1 and blue lines) at time $t = 0$. It proposes a block B_1 extending the genesis block since it is the first proposer, but generally the servers extend the block proposed by the leader for the previous round. At time $t = \delta$, p_2 proposes the next block B_2 immediately after receiving the block B_1. Note that this gives us $\gamma \leftarrow \kappa + 1$ except with negligible probability.

A Byzantine leader can try to slow down the protocol or may not send its proposal to the next leader. To ensure that a correct leader is always able to propose, all correct servers also forward the proposals of the current round to the next leader (Step 2 and gray lines).

The Propose Step and the Relay Step follow each other with different leaders drawn from $\mathcal{P}$. Additionally, in every round, the correct servers commit blocks after removing the top κ blocks from their local chains.

4.2 Handling Faults

Next, we give an intuition of fault-handling in Apollo and present a concise technical description in Fig. 3.

Let L_r be the leader of round r. Let κ be the security parameter.

1. **Propose.** On receiving block B_{k-1} for round $r-1$, the leader L_r for round r proposes a block $B_r := \langle h_k, \mathsf{cmds} \rangle_L$ 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}, set timer $\mathsf{blTimer}(r+1)$ to 4Δ and start counting down (refer Fig. 3). Cancel all timers for lower rounds.
3. **(Non-blocking) Commit.** On receiving a *valid* chain of blocks $\mathcal{C} := \{B_0, \ldots, B_\ell\}$ commit blocks $\mathcal{C}[: \ell - \kappa]$ if $\ell > \kappa$.

Fig. 2: **Rounds in Apollo protocol.**

Block Equivocation. A leader can equivocate by sending different blocks to different correct servers. 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 servers in two ways: (1) A correct server whose head of the chain is $B_{k'}$ obtains 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 server's local chain. If L_r cannot provide valid ancestors within 2Δ, then the correct server blames L_r by sending a blame message. A correct L_r can always respond to such queries and therefore not get blamed by correct servers. When the parent block is received, a correct server may realize equivocations due to conflicts with the local chain. (2) A correct server gets two different blocks during the *Relay* Step (Step 2). In both of these cases, the correct servers multicast the equivocations to all the other servers if it detected it directly, or via forwarding the blame from others. All the correct servers include the equivocation blame as a meta-transaction in their future proposals until it is committed.

It is not secure to update the proposer set or punish the Byzantine server on obtaining a blame certificates, until it is committed. This is because we do not use timing guarantees, i.e., rely on Δ, to ensure that all correct servers have detected and agreed on the equivocation. On committing a block containing the blame certificates, we know that sufficient correct servers have extended the block, thereby ensuring all the other correct servers will learn about the Byzantine server.

Crashed Leader(s). Consider leaders L_r and L_{r+1} for rounds r and $r+1$ respectively. Let L_r not propose any block. Now, the correct servers could be processing/waiting for blocks at different rounds $\leq r-1$. The first correct server to finish processing the block B_k for round $r-1$, will wait for 4Δ time (we will describe soon why to wait for 4Δ) after relaying B_k before blaming L_r. Upon timing out, a correct server cannot be sure if all the correct servers are waiting for a proposal for round r, since our protocol can proceed at network speed. Therefore, different correct servers could be waiting for blocks from rounds $r' < r$. This case can also occur if the Byzantine leaders send the proposals to

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

1. **Blames.** The server p_i always does the following:
– **No-progress Blame.** If $\mathsf{blTimer}(r)$ expires 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 Fig. 2. Multicast the certificate $\mathcal{C}(r, \mathsf{NPBlame})$ to all the servers.
– **Equivocation.** If there exists two valid blocks B and $B^\star$ proposed by server $p_j \in \mathcal{N}$ in any round $r^\star$ for the first time, 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 blame certificates $\langle r, \mathsf{EQBlame}\rangle_i$ or $\mathcal{C}(r, \mathsf{NPBlame})$, remove the leader from the proposer set $\mathcal{P}$.

Fig. 3: **Handling Byzantine behavior in Apollo protocol.**

some correct servers, who will then be ahead in round number when compared to the servers that did not receive the proposals.

In any case, a correct server on timeout for round r, sends the latest block B_k to all the correct servers in order to synchronize all the correct servers up to round $r-1$. This multicast is not done during the steady state in order to obtain the desired linearity in the steady state as this step has a communication complexity of $O(n^2)$ when $n-f$ servers time out. We know that within another 4Δ all servers will relay the proposal to L_r and then blame when it does not respond. On collecting $n/2+1$ such blame messages, all the correct servers build a virtual block for round 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.

Round-Relative Timers. Apollo uses *round-relative* timers, i.e., blame based on the latest round. Earlier works [3, 36] use stable leaders and *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 [3] and OptSync [36], the servers needlessly wait for 2004Δ before blaming the leader. We overcome this, since our timers are always rooted at the last received block.

Why is 4Δ timeout sufficient? Say p_i is the first server that enters round r at time t. It relays the previous block to the current leader L_r which will reach L_r by time $t+\Delta$. A correct leader L_r may not recognize this chain, and request the full chain. This request will reach p_i by time $t+2\Delta$. A correct leader will then immediately propose since it has a valid chain to extend. This proposal will reach p_i by time $t+4\Delta$. Therefore, waiting for a total of 4Δ after relaying the block is always sufficient for a correct server to propose, and thus ensure that a correct leader is never blamed a correct server.

Security Analysis. Due to lack of space, we state the security theorems here without proofs and defer the security analysis to Appendix B.

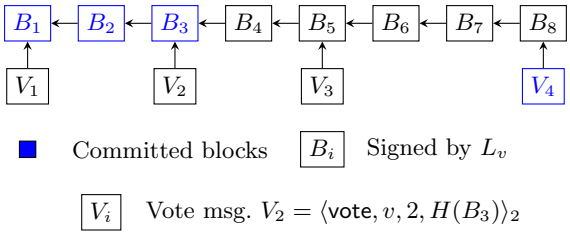

Fig. 4: **High-level overview of Artemis (n = 5).** V_i are vote messages proposed by servers p_i. We apply UCR using the vote messages resulting in block commits (in blue).

Theorem 2 (Apollo Safety). *For any height $k \geq 0$, if two correct servers commit to blocks B and $B^\star$, then $B = B^\star$.*

Theorem 3 (Apollo Liveness). *Assuming standard synchrony, Apollo always makes progress, and commits blocks with a period of at most 12Δ.*

5 Artemis Protocol

Round-robin protocols working at the network speed can be slower than stable leader protocols since a slow leader can slow the system, while a stable leader may make faster progress as evidenced in practical implementations [26]. In this section, we construct Artemis which uses a view leader that coordinates the chain, and still allows applying UCR by running a Apollo sub-protocol on the chain produced by the view leader.

Views. In stable-leader SMRs [3, 8, 13, 36, 36, 43], a view number v represents a period with a stable leader L_v. A change in view number indicates a leader change. In Artemis, a view number denotes a period with a fixed view leader.

Artemis uses two leaders: view *leader* L_v and *round leader* L_r. The view leader L_v of view v creates *blocks* and builds a chain. The round leader L_r for round r runs Apollo sub-protocol by creating proposals called *votes* containing the hash of the latest block from the view leader.

5.1 Steady-state protocol

An overview is presented in Fig. 4. The blocks B_1 to B_8 are signed by the view leader L_v. The vote messages $V_i = \langle \mathsf{vote}, v, r, H(B) \rangle_i$ are signed by p_i. Intuitively, we can visualize Artemis as using L_v to build a chain of blocks, and simultaneously using Apollo on vote messages using round leaders L_r. The vote messages form a tamper-resistant chain, and the round leaders are chosen akin to Apollo.

The view leader L_v of view v collects transactions from the clients and creates a chain of blocks. Like related works [3, 15, 16, 36, 43], we assume that there

Let L_v be the view leader for the view v.
View Leader. L_v creates blocks $B_k := \langle b_k, H(B_{k-1})\rangle_{L_v}$ and multicasts the chain of blocks to all the servers. The servers p_i do the following in round r:

1. **Update chain.** On receiving valid blocks from L_v add them to the locally stored chain. Set $\mathsf{blameViewLeader} \leftarrow \Delta$.
2. **Vote.** If $p_i = L_r$ for the round r in view v, then it multicasts $\langle \mathsf{vote}, v, r, H(B_k)\rangle_i$ where B_k is the highest height block known to the server.
3. **Relay.** On receiving $\langle \mathsf{vote}, v, r, H(B_k)\rangle_{L_r}$ for a valid chain containing B_k, forward the vote message to L_{r+1} along with the latest known block B_k. Set $\mathsf{blTimer}(r+1)$ to 4Δ and cancel all timers for rounds $r' \leq r$.

(Non-blocking) Commit. On receiving a *valid* chain of votes $\mathcal{C}^\star := \{V_0, \ldots, V_\ell\}$ commit all ancestors of the block referred by the vote $V_{\ell-\kappa}$ if $\ell > \kappa$.

Fig. 5: **Steady-state protocol for Artemis.**

are always sufficient transactions available[11]. Thus, every server must receive p blocks in $p\Delta$ time from L_v. Due to the synchrony assumptions, if block B_k is received at time t, then B_{k+1} must be received within time $t + \Delta$. This is because L_v does not need any interaction to create blocks in the steady-state, unlike related works [3,5,36,43] which requires every block to contain a quorum certificate and thus requiring a round-trip of communication.

The round-leaders can be viewed as running Apollo sub-protocol. A vote is a block for the Apollo sub-protocol. A series of vote messages for consecutive rounds forms a vote chain. A vote at round r for block B_k is connected indirectly to the vote at round $r+1$ for block B_{k+1} via the hash pointers between B_k and B_{k+1} ensuring tamper-resistance.

A key difference between blocks in Apollo and in Artemis is that vote messages can produce and commit multiple blocks between each proposal. When the view leader is correct, Byzantine servers do not affect the throughput of the system, as the fast correct servers will collect more blocks while the Byzantine servers slow the system down, and include the highest hash in its turn to send the vote message. This will eventually result in committing a large volume of transactions. In the example illustration in Fig. 4, the vote message V_4 results in the block B_3 and its ancestors having $\kappa + 1 = f + 1 = 3$ children and hence results in committing them.

Artemis retains the round-relative blaming property for the view leader from Apollo which improves the worst-case performance of Artemis over the state-of-the-art related works [36].

Let L_v and L_{v+1} be the leaders for views v and $v+1$. The server p_i does the following:

- **Round Leader Equivocation.** If two equivocating votes are observed from a server p_j directly or indirectly, then multicast $\langle \mathsf{Blame}, j \rangle_i$, the equivocating votes, and the latest vote received.
- **Slow Round Leader.** If $\mathsf{blTimer}(r)$ expires for a round r, multicast $\langle \mathsf{Blame}, L_r \rangle_i$ and the latest vote message received.
- **Blame Certificates for Round Leaders.** On collecting $n/2+1$ blame messages or a blame certificate for server p_j, multicast the blame certificate and include it in the next vote message. On committing a block with vote containing a blame certificate, optionally remove p_i from $\mathcal{P}$.
- **View Leader Equivocation.** If two equivocating blocks are observed from L_v directly or indirectly, then multicast $\langle \mathsf{Blame}, L_v, v \rangle_i$ and the equivocating blocks. Quit view v and stop processing messages in view v.
- **Slow View Leader.** If $\mathsf{blameViewLeader}$ expires, multicast $\langle \mathsf{Blame}, L_v, v \rangle_i$.
- **Quit View.** On receiving $f+1$ $\langle \mathsf{Blame}, L_v, v \rangle$ messages or $\mathcal{C}(\mathsf{Blame}, L_v, v)$, quit the view v if we haven't quit already. Multicast $\mathcal{C}(\mathsf{Blame}, L_v, v)$, and wait Δ to enter view $v+1$.
- **New View.** Perform the following steps:
  (i) Lock on to the block B_{lck} referred by the highest vote in view v. Send the head of the local chain and the highest vote to L_{v+1} and wait for 5Δ for the first block in view $v+1$.
  (ii) L_{v+1} waits for 2Δ and requests missing blocks (if any), and then sends $\langle B_k, v+1, V_r \rangle_{L_v}$ where B_k is a block extending the highest block from view v, V_r is the highest view v vote received. Collect $f+1$ votes, and include $\mathcal{C}(\mathsf{newView}, v+1, H(B_k))$ in the next block.
  (iii) For the first block in view $v+1$ send $\langle \mathsf{newView}, v+1, H(B_k) \rangle_i$ if B_k extends B_{lck} block (equal to or longer than). Send $\langle \mathsf{newView}, v+1, H(B_k) \rangle_i$ to L_{v+1} and wait for 4Δ. Ensure that the next block contains $\mathcal{C}(\mathsf{newView}, v+1, H(B_{k-1}))$ and continue the steady-state.

Fig. 6: **Handling Byzantine faults in Artemis.**

5.2 Handling Byzantine Behavior

Fig. 6 describes the protocol to handle Byzantine behavior in Artemis. We discuss three states: (i) L_v is correct, (ii) L_v crashes, and (iii) L_v is Byzantine.
Case (i). When L_v is correct, the vote chain is exactly like Apollo except that the blocks are detached from the proposals and come from L_v. A Byzantine server cannot forge alternate blocks, and can thus only crash or send messages slowly. In the former case, we simply blame the server and use a blame certificate as a virtual block for round r. The latter case does not affect throughput as other correct servers will keep downloading the chain and proposing them during their turn to propose, thereby ensuring all the servers catch-up to the latest chain.

[11] This assumption can be removed by slightly changing the blaming mechanism to not blame if the local transaction buffer is empty and attempting to send transactions to L_v on timeout first, and then blaming. An example of this implementation can be found in Concord-BFT [23].

Case (ii). When L_v crashes at time t, the round-relative timer kicks in for all correct servers by time $t + \Delta$. By time $t + 2\Delta$, all the correct servers will blame and by $t + 3\Delta$ obtain $\mathcal{C}(\mathsf{Blame}, L_v, v)$, and thus quit the view v. By time $t + 4\Delta$ all the correct servers enter the view $v + 1$. The extra Δ wait is used to ensure that all servers stop processing vote messages in the view v.

In the new view $v + 1$, all the correct servers lock on to the highest block known from the view v. From the safety properties of Apollo, we know that if a correct server commits a block, then all possible chains must extend the committed block. The servers then send their highest view v block to L_{v+1}. By time $t + 5\Delta$, L_{v+1} receives all the chains, and by time $t + 7\Delta$, L_{v+1} has all the chains and proposes the first block for the view $v+1$ which will reach all correct servers by time $t+8\Delta$ well within the 5Δ timer for the first block from L_{v+1} by the other correct servers. We ensure that the second block in the view $v+1$ must be certified, which guarantees that the chain selected by L_{v+1} extends at least one honest server's locked block and has provided a convincing vote message from view v. The latter guarantees that the highest committed block must be extended in the new view $v + 1$.

Case (iii). If L_v is Byzantine and tries to equivocate (via vote message or directly sending B_k and $B_k^\star$ to different servers), it will be detected by all correct servers within $O(\kappa\Delta)$ and thus trigger view-change. This does not affect the safety as at least one vote in the vote chain is from a correct server whose vote pins the blocks at that height, akin to Apollo.

During the view-change, Byzantine servers may add votes with the help of L_v and send them to only some correct servers. In the worst case, Byzantine servers can add up to f more vote messages. This is not a concern, since all the servers lock on to blocks that extends the highest known vote message. Thus, any chain in the next view $v + 1$ is guaranteed to result in committing of the highest committed block in view v.

Security Analysis. Using Theorem 4 and Theorem 5 we prove that Artemis is a secure SMR protocol in standard synchrony. See Appendix C for proofs.

Theorem 4 (Artemis Safety). *If two correct servers commit blocks B_k and $B_k^\star$ at height k, then $B_k = B_k^\star$.*

Theorem 5 (Artemis Liveness). *Assuming standard synchrony, Artemis always makes progress, and commits blocks with a period of at most $O(\kappa\Delta)$.*

6 Publicly Verifiable SMR

A publicly verifiable SMR allows the clients to verify the state of the SMR protocol without having to contact the servers that run the protocol. Prominent blockchain protocols [10,33,40] are naturally publicly verifiable as their commit rules are properties of their chains[12]. However, not all protocols can use the

[12] In Proof-of-Stake protocols, the stake is defined by the chain, and thus the leaders are publicly verifiable. However, the public verifiability of the chain depends on the underlying SMR used in the protocol.

quorum certificates generated for agreement to convince the clients that it is the accepted block. For instance, in Sync-HotStuff [3], multiple quorum certificates could be generated for a round without the correct servers having heard them. By contacting a Byzantine server, a correct client can be convinced of an incorrect state in this manner.

Permissioned protocols [3, 13, 36] can be made publicly verifiable (if not already) by building quorum certificates on the state for every height. The clients can then verifiably obtain the state using the quorum certificate. For a chain of length ℓ, this incurs a signature complexity of $O(\ell f)$ for the SMR servers.

We can use UCR to design an improved protocol to ensure public verifiability for any SMR. Intuitively, we can run Apollo (Fig. 1) protocol with the SMR commits as input without the fault tolerance. The liveness property of the underlying SMR automatically guarantees progress. This results in a signature complexity for a chain of length ℓ to $O(\ell + \kappa)$.

Consider any SMR protocol that implements Definition 1 with n servers tolerating f faults. Then every log position $r > 0$ has some state S_r attached to it. Committing blocks can be observed as agreement on S_r, and clients of SMR protocols need S_{latest}, where, the latest means that a state that can only be up to Δ time old for synchronous systems.[13]

We present a simple UCR-based publicly verifiable SMR protocol in Fig. 7 that is agnostic to the underlying details of the SMR. In this new protocol, we first chain the states together by including the hash of the parent state using state tuples $v_r = \langle H(S_{r-1}), S_r \rangle$ for the log position r. A designated server with id $i = H(R, r) \mod n$ signs and sends this to all the active clients. We can also use gossip networks to diffuse this message as done by Bitcoin [33].

A client that obtains any $\kappa + 1$ such signed v_j for $j > i$, and downloads the chain, both of which can come from any external source, can be guaranteed that S_r must be committed (see Lemma 1). Note that, unlike Apollo, here we do not blame the server with id i if no v_r are received. The protocol can still continue because the hash chain and $\kappa + 1$ implicit votes guarantee that no other root state S_r can get committed.

Servers. For every round r, if $H(R, r) \mod n = j$, then the server p_j gossips $v_r := \langle H(S_{r-1}), S_r \rangle_j$ (or some digest of S_r).
Clients. Any external client commits S_r if all the following are satisfied: (a) collects valid $(v_{r_1}, v_{r_2}, \ldots, v_{r_f})$, (b) $r \leq r_1 < r_2 < \cdots < r_{\kappa+1}$, (c) obtains $H(S_{r_i}), H(S_{r_{i-1}})$ for $r_1 \leq i \leq r_{\kappa+1}$, and (d) downloads S_r. Here, by valid we mean that all the messages are correctly signed.

Fig. 7: Publicly verifiable SMR.

[13] We cannot discuss it in terms of block heights because any number of blocks might be successfully committed within Δ because of the responsiveness of our protocols. For partially synchronous systems it is not possible to guarantee any form of the latest state before GST.

To ensure safety, we show in Lemma 1 that the commits made by any client with access to information about the chain and v_i must be the same as the state committed by the correct servers in the underlying SMR protocol. We state the security lemma here and defer the formal proof to Appendix A.

Lemma 1 (Commit Safety). *If a correct client commits S_r using Fig. 7 for round r, then all the correct SMR servers must have committed S_r.*

7 Acknowledgements

We thank Ling Ren and Ittai Abraham for helpful feedback on the various applications of UCR, Kartik Nayak for discussions regarding the good-case latency, Nibesh Shrestha for feedback on the draft, and Manish Nagaraj for early discussions. This work was supported in part by NIFA award number 2021-67021-34252 and by the National Science Foundation (NSF) under grant CNS1846316.

References

1. Abraham, I., Devadas, S., Dolev, D., Nayak, K., Ren, L.: Synchronous Byzantine Agreement with Expected O(1) Rounds, Expected $$O(n\hat{2})$$ Communication, and Optimal Resilience. In: Goldberg Ian, and Moore, T. (eds.) Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 11598 LNCS, pp. 320–334. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-32101-7_20, http://link.springer.com/10.1007/978-3-030-32101-7_20
2. Abraham, I., Malkhi, D., Nayak, K., Ren, L.: Dfinity Consensus, Explored. IACR Cryptology ePrint Archive **2018**, 1153 (2018), https://eprint.iacr.org/2018/1153
3. Abraham, I., Malkhi, D., Nayak, K., Ren, L., Yin, M.: Sync HotStuff: Simple and Practical Synchronous State Machine Replication. In: 2020 IEEE Symposium on Security and Privacy (SP). vol. 2020-May, pp. 106–118. IEEE, Oakland (May 2020). https://doi.org/10.1109/SP40000.2020.00044, https://ieeexplore.ieee.org/document/9152792/
4. Abraham, I., Nayak, K., Ren, L., Xiang, Z.: Good-case Latency of Byzantine Broadcast. In: Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing. pp. 331–341. ACM, New York, NY, USA (Jul 2021). https://doi.org/10.1145/3465084.3467899, https://dl.acm.org/doi/10.1145/3465084.3467899
5. Abraham, I., Nayak, K., Shrestha, N.: Optimal Good-Case Latency for Rotating Leader Synchronous BFT. In: Bramas, Q., Gramoli, V., Milani, A. (eds.) 25th International Conference on Principles of Distributed Systems (OPODIS 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 217, pp. 27:1–27:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). https://doi.org/10.4230/LIPIcs.OPODIS.2021.27, https://drops.dagstuhl.de/opus/volltexte/2022/15802
6. Baudet, M., Ching, A., Chursin, A., Danezis, G., Garillot, F., Li, Z., Malkhi, D., Naor, O., Perelman, D., Sonnino, A.: State machine replication in the libra blockchain (2019), https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2020-05-26.pdf

7. Boneh, D., Lynn, B., Shacham, H.: Short Signatures from the Weil Pairing. Journal of Cryptology **17**(4), 297–319 (Sep 2004). https://doi.org/10.1007/s00145-004-0314-9, https://doi.org/10.1007/s00145-004-0314-9
8. Buchman, E., Kwon, J., Milosevic, Z.: The latest gossip on bft consensus (2019)
9. Buchman, E., Kwon, J., Milosevic, Z.: The latest gossip on BFT consensus pp. 1–14 (Nov 2019), http://arxiv.org/abs/1807.04938
10. Buterin, V., Griffith, V.: Casper the friendly finality gadget (2019)
11. Cachin, C., Kursawe, K., Shoup, V.: Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. Journal of Cryptology **18**(3), 219–246 (Jul 2005). https://doi.org/10.1007/s00145-005-0318-0, http://link.springer.com/10.1007/s00145-005-0318-0
12. Castro, M., Liskov, B.: Practical byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems (TOCS) **20**(4), 398–461 (2002)
13. Castro, M., Liskov, B.: Practical byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems **20**(4), 398–461 (Nov 2002). https://doi.org/10.1145/571637.571640, https://doi.org/10.1145/571637.571640
14. Chan, B.Y., Shi, E.: Streamlet: Textbook Streamlined Blockchains. In: Proceedings of the 2nd ACM Conference on Advances in Financial Technologies. pp. 1–11. AFT '20, Association for Computing Machinery, New York, NY, USA (Oct 2020). https://doi.org/10.1145/3419614.3423256, https://doi.org/10.1145/3419614.3423256
15. Chan, T.H.H., Pass, R., Shi, E.: Pala: A simple partially synchronous blockchain. IACR Cryptol. ePrint Arch. **2018**, 981 (2018)
16. Chan, T.H.H., Pass, R., Shi, E.: Pili: An extremely simple synchronous blockchain. IACR Cryptol. ePrint Arch. **2018**, 980 (2018)
17. github - vmware/concord-bft: concord byzantine fault tolerant state machine replication library 2021 (2021), https://github.com/vmware/concord-bft
18. Danezis, G., Kogias, E.K., Sonnino, A., Spiegelman, A.: Narwhal and Tusk: A DAG-based Mempool and Efficient BFT Consensus, vol. 1. Association for Computing Machinery (2021). https://doi.org/10.1145/3492321.3519594, http://arxiv.org/abs/2105.11827
19. David, B., Gaži, P., Kiayias, A., Russell, A.: Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). vol. 10821 LNCS, pp. 66–98 (2018). https://doi.org/10.1007/978-3-319-78375-8_3, https://link.springer.com/chapter/10.1007/978-3-319-78375-8_3
20. Duan, S., Meling, H., Peisert, S., Zhang, H.: BChain: Byzantine Replication with High Throughput and Embedded Reconfiguration. In: Aguilera, M.K., Querzoni, L., Shapiro, M. (eds.) Principles of Distributed Systems. pp. 91–106. Lecture Notes in Computer Science, Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-14472-6_7
21. Gelashvili, R., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A., Xiang, Z.: Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback (Jun 2021), http://arxiv.org/abs/2106.10362
22. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In: SOSP 2017 - Proceedings of the 26th ACM Symposium on Operating Systems Principles. pp. 51–68. ACM, New York (2017). https://doi.org/10.1145/3132747.3132757

23. Golan Gueta, G., Abraham, I., Grossman, S., Malkhi, D., Pinkas, B., Reiter, M., Seredinschi, D.A., Tamir, O., Tomescu, A.: SBFT: A Scalable and Decentralized Trust Infrastructure. In: 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 568–580. IEEE (Jun 2019). https://doi.org/10.1109/DSN.2019.00063, https://ieeexplore.ieee.org/document/8809541/
24. Guo, Y., Pass, R., Shi, E.: Synchronous, with a Chance of Partition Tolerance. In: Boldyreva, A., Micciancio, D. (eds.) Advances in Cryptology – CRYPTO 2019. pp. 499–529. Lecture Notes in Computer Science, Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-26948-7_18
25. Hanke, T., Movahedi, M., Williams, D.: Dfinity technology overview series, consensus system (2018)
26. Hot-Stuff: hot-stuff/libhotstuff (2021), https://github.com/hot-stuff/libhotstuff
27. Keidar, I., Kokoris-Kogias, E., Naor, O., Spiegelman, A.: All You Need is DAG. In: Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing. pp. 165–175. ACM, New York, NY, USA (Jul 2021). https://doi.org/10.1145/3465084.3467905, https://dl.acm.org/doi/10.1145/3465084.3467905
28. Keidar, I., Naor, O., Shapiro, E.: Cordial Miners: Blocklace-Based Ordering Consensus Protocols for Every Eventuality (Aug 2022). https://doi.org/10.48550/arXiv.2205.09174, http://arxiv.org/abs/2205.09174
29. Lamport, L., Shostak, R., Pease, M.: The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems **4**(3), 382–401 (Jul 1982). https://doi.org/10.1145/357172.357176, https://dl.acm.org/doi/10.1145/357172.357176
30. Malkhi, D., Nayak, K., Ren, L.: Flexible byzantine fault tolerance. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. p. 1041–1053. CCS '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3319535.3354225, https://doi.org/10.1145/3319535.3354225
31. Malkhi, D., Szalachowski, P.: Maximal Extractable Value (MEV) Protection on a DAG (Sep 2022), http://arxiv.org/abs/2208.00940
32. Momose, A., Ren, L.: Multi-Threshold Byzantine Fault Tolerance. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. pp. 1686–1699. CCS '21, Association for Computing Machinery, New York, NY, USA (Nov 2021). https://doi.org/10.1145/3460120.3484554, https://doi.org/10.1145/3460120.3484554
33. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Tech. rep., Manubot (2019)
34. ConsenSys/quorum (Sep 2021), https://github.com/ConsenSys/quorum, original-date: 2016-11-14T05:42:57Z
35. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys **22**(4), 299–319 (Dec 1990). https://doi.org/10.1145/98163.98167, https://dl.acm.org/doi/10.1145/98163.98167
36. Shrestha, N., Abraham, I., Ren, L., Nayak, K.: On the Optimality of Optimistic Responsiveness. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. pp. 839–857. ACM, New York, NY, USA (Oct 2020). https://doi.org/10.1145/3372297.3417284, https://dl.acm.org/doi/10.1145/3372297.3417284

37. Spiegelman, A., Giridharan, N., Sonnino, A., Kokoris-Kogias, L.: Bullshark: DAG BFT Protocols Made Practical. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 2705–2718. CCS '22, Association for Computing Machinery, New York, NY, USA (Nov 2022). https://doi.org/10.1145/3548606.3559361, https://doi.org/10.1145/3548606.3559361
38. Team, T.D.: The Internet Computer for Geeks (2022), https://eprint.iacr.org/2022/087
39. Tendermint: tendermint/tendermint: Tendermint core (bft consensus) in go, https://github.com/tendermint/tendermint
40. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)
41. Xiang, Z., Malkhi, D., Nayak, K., Ren, L.: Strengthened Fault Tolerance in Byzantine Fault Tolerant Replication. In: 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS). pp. 205–215 (Jul 2021). https://doi.org/10.1109/ICDCS51616.2021.00028
42. Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: Hotstuff: Bft consensus with linearity and responsiveness. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. pp. 347–356. ACM (2019)
43. Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: HotStuff: BFT Consensus with Linearity and Responsiveness. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. pp. 347–356. PODC '19, Association for Computing Machinery, New York, NY, USA (Jul 2019). https://doi.org/10.1145/3293611.3331591, https://doi.org/10.1145/3293611.3331591

A Security Analysis

Lemma 2 (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* $\ell > 0$*. If* $B_\ell = B_\ell^\star$*, then* $B_{\ell-1} = B_{\ell-1}^\star$*, …,* $B_0 = B_0^\star$*, i.e.,* $\mathcal{C} = \mathcal{C}^\star$*.*

Proof. Given a valid chain $\{B_0, \ldots, B_{k-1}, B_k, B_{k+1}, \ldots, B_\ell\}$ and for any $k \leq \ell$, it is *not possible* (except with negligible probability) to obtain another valid chain of blocks $\{B_0, \ldots, B_{k-1}, B_k^\star, B_{k+1}, \ldots, B_\ell\}$ where $B_k \neq B_k^\star$, since changing a block naturally changes its hash thereby rendering all its direct and indirect children invalid members of the tampered chain. □

Proof of Theorem 1. We restate Theorem 1 here for convenience.

Theorem 1 (Unique Chain Rule). *Consider a protocol for* n *servers tolerating* f *Byzantine faults, and satisfying Definition 2. Then, on observing a valid chain* $\mathcal{C} := \{B_0, \ldots, B_\ell\}$ *of size* ℓ *(with* $\ell > \gamma$*), commit the prefix chain* $\mathcal{C}[: \ell - \gamma]$*.*

Proof. Suppose for the sake of contradiction such a protocol outputs two chains $\mathcal{C} := \mathcal{C}_P || \mathcal{C}_S$ and $\mathcal{C}' := \mathcal{C}'_P || \mathcal{C}'_S$ where $\mathcal{C}_P \cap \mathcal{C}'_P \neq \emptyset$. Since $\mathcal{C}_P \cap \mathcal{C}'_P \neq \emptyset$ there exists at least one block that differs in these two chains. Therefore, from Lemma 2 $\mathcal{C}_S \neq \mathcal{C}'_S$, i.e., no block is the same.

However, both of these prefixes contain blocks from $f+1$ correct servers. Since $n = 2f + 1$, either there exists two correct servers in $\mathcal{C}_S$ and $\mathcal{C}'_S$ that does not extend the block proposed by the other server in $\mathcal{C}'_S$ and $\mathcal{C}_S$ respectively, leading to a contradiction that there are only f Byzantine servers, or there is a common block in $\mathcal{C}_S$ where both of these correct servers extend each other's blocks, which leads to a contradiction with $\mathcal{C}_S \neq \mathcal{C}'_S$ or the violation of Lemma 2. □

Proof of Lemma 1.

Lemma 1 (Commit Safety). *If a correct client commits S_r using Fig. 7 for round r, then all the correct SMR servers must have committed S_r.*

Proof. If a correct client commits S_r, then Step (a), Step (b), and Step (c) must be satisfied. From Step (c) we know that the chain from S_{r_1} to $S_{r_{f+1}}$ is tamper resistant, and from Step (a) and Step (b) we know that at least one correct server attests to the fact that one of $\{S_r, \ldots, S_{r_{f+1}}\}$ is committed. Since they are tamper resistant, S_r must be committed by a correct server. From the security of Definition 1, we know that if a correct server commits S_r then all the other correct servers commit S_r. □

B Security Analysis for Apollo

In this section, we prove safety and liveness of Apollo in standard synchrony. The commit rule employed by Apollo from Theorem 1 guarantees that no two conflicting blocks will be committed by any correct server. On a high-level, we want the following important properties to ensure security: (i) Correct leaders are always able to propose, and (ii) Correct proposals are always committed.

Definition 3 (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 the following constraints to chain validity (in addition to the definitions from Section 2):

Definition 4 (Valid Chain). *A valid chain $\mathcal{C} := \{B_0, \ldots, B_\ell\}$ of size ℓ consists of:*

– A hash chain of blocks (from Section 2), i.e., for any $0 \leq i < \ell$, 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 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 servers can crash/equivocate. From the tamper resistance property (Lemma 2),

we are guaranteed that a Byzantine server 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 server will always be committed.

Theorem 6 (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 servers 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 $n/2 + 1$ blames against the correct leader.

Let us prove this by induction. The base case holds trivially for $r = 0$. Assume it is true up to round $r - 1$. Let p_i be the earliest correct server to reach round r, either via a timeout 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 servers can be 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 servers with a block B_k for round r by time $t+4\Delta$ (after requesting any unknown chain). Since the earliest correct server does not timeout, no other correct server can timeout for L_r. Therefore, the statement holds true for round r.

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 $\kappa+1$ long, when $L_{r+\kappa+1}$ proposes a block. This ensures that B is committed. □

Now, we prove the safety of the commit rule.

Proof of Theorem 2.

Theorem 2 (Apollo Safety). *For any height $k \geq 0$, if two correct servers 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 servers 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) chains $\{B_k, \ldots, B_{k+\kappa}\}$ and $\{B^\star_k, \ldots, B^\star_{k+\kappa}\}$. From the tamper resistance property (Lemma 2), this implies that there are two proposals for *all* blocks starting from height k to $k+\kappa$. This implies $\kappa+1$ servers equivocated or have blame certificates. Since each leader is chosen randomly (with or without replacement), the probability of **all** $\kappa+1$ leaders being Byzantine is negligible in κ. Therefore, we cannot obtain two chains as described previously. □

To prove liveness, from Theorem 6, we know that in one of the servers in rounds $k^\star \in \{k, \ldots, k+\kappa\}$ is correct, and therefore its block is eventually (by

round $k^\star + \kappa + 1$) committed, thereby committing B. We prove it formally in Theorem 3.

Proof of Theorem 3.

Theorem 3 (Apollo Liveness). *Assuming standard synchrony, Apollo always makes progress, and commits blocks with a period of at most* 12Δ.

Proof. Let t be the current global clock time, at which time the first correct server reaches round r. It enters round r by obtaining a blame certificate or a block for round $r-1$. In the worst case, due to the network delay or due to Byzantine servers, a correct server will blame a crashed leader L_r for round r at time $t + 4\Delta$, followed by multicasting the block to all the servers. This block and blame will reach all the correct servers by time $t + 5\Delta$. At time $t + 7\Delta$, all the correct servers will enter round r if not already in round r.

Now all these correct servers multicast the block for round $r-1$ to the crashed leader L_r again and wait for 4Δ before sending a blame at time $t+11\Delta$. This blame takes an additional Δ time to reach all other correct servers by time $t + 12\Delta$. At this point, we commit one block, because this virtual blame certificate block extends the local chain of every correct server. Therefore, in the worst case, it can take 12Δ for a block to be committed, but we will always make progress.

The 12Δ wait only occurs f times in the lifetime of networks assuming standard synchrony. If all the servers are correct, but the network is slow, then we always progress with a period of Δ. □

Theorem 2 and Theorem 3 prove that Apollo is a secure SMR protocol under the standard synchrony assumption.

C Security Analysis of Artemis

In this section, we prove the security of Artemis in standard synchrony. On a high level, we want to guarantee the following to ensure *safety* and *liveness*:

(i) If two correct servers commit B_k and $B_k^\star$, for the same height k and view v, then $B_k = B_k^\star$.
(ii) If two correct servers commit B_k and $B_k^\star$ for the same height k but across different views, then $B_k = B_k^\star$.
(iii) A correct view leader can always propose blocks and will never be blamed by correct servers.
(iv) A correct round leader can always vote.

Lemma 3 (Safety within a view). *If two servers commit blocks* B_k *and* $B_k^\star$ *at height* k *in the same view* v*, then* $B_k = B_k^\star$.

Proof. The proof is analogous to the proof from Apollo except that the blocks come from the view leader L_v. The proof still holds even if L_v equivocates, or crashed but the view change has not occurred yet. □

Lemma 4. *During a view change from view v to view $v+1$, a correct view leader L_{v+1} will not be blamed by any correct server.*

Proof. Let B_k is the highest committed block by some correct server p_i in view v. This implies that there exist V_j through $V_{j+\kappa}$ that resulted in committing B_k. We know that at least one of these proposers are correct with non-negligible probability and thus irreplaceable. Let T be the time at which the earliest server p_m enters view $v+1$ by obtaining the blame certificate, multicasting it and waiting for Δ. By time $T+\Delta$, all correct servers will enter the view $v+1$. By the end of time $T+\Delta$, all the correct servers will lock on to their highest known chain, and send the highest vote message and the highest block to the next leader L_{v+1}. By time $T+2\Delta+2\Delta = T+4\Delta$ (additional 2Δ to download the latest chain), L_{v+1} obtains a chain and a vote message that is at least as high as the highest among all the correct servers. The leader will now pick the highest chain, and extend it to create the first block in view $v+1$ which will reach all the correct servers by time $T+5\Delta$. Thus, no correct server will blame L_{v+1} because of the 5Δ timeout for the first block.

Since the new view leader L_{v+1} is correct, and it extended the longest chain and also send the longest vote which extends the lock of all the correct servers, all the correct servers will send the message $\langle \mathsf{newView}, v+1, \cdot \rangle$ by time $T+5\Delta+2\Delta$ or $T+7\Delta$ (the additional 2Δ for the correct servers (non-leaders) to download the new chain). These messages will reach L_{v+1} by time $T+8\Delta$ at which point the leader constructs the second block along with the certificate and multicasts it which will reach all the correct servers by time $T+9\Delta$. Thus, no correct server will blame L_{v+1} because of a timeout for the second block. □

Lemma 5 (Unique Extensibility across views). *Let B_k be the highest block committed by a correct server in a view v. Let $v' > v$ be any view that comes after the view v. Then all valid chains in view v' must extend B_k.*

Proof. Consider $v' = v+1$. A valid chain formed in view $v+1$, must have obtained a certificate $\mathcal{C}(\mathsf{newView}, v+1, \cdot)$. This implies at least one correct server voted to validate the new view which implies that L_{v+1} produced a valid $V\star$ message that extends the lock of this correct server. By the safety properties of Apollo, $V\star$ must extend B_k since the fact that B_k was committed, implies that there is a sequence of $f+1$ votes that extend B_k and thus $V\star$ must also extend B_k. □

Lemma 6 (Safety across views). *If two servers commit blocks B_k and $B_k^\star$ across views v and $v' > v$, then $B_k = B_k^\star$.*

Proof. Let p_i and p_j commit B_k and $B_k^\star$ in view v and v' respectively. From Lemma 5, we know that all valid chains in view v' must extend B_k which leads to a contradiction. Therefore, $B_k^\star = B_k$ must be true. □

Finally, we prove the safety of Artemis.

Theorem 4 (Artemis Safety). *If two correct servers commit blocks B_k and $B_k^\star$ at height k, then $B_k = B_k^\star$.*

Proof. If two correct servers commit B_k and $B_k^\star$ in the same view, then from Lemma 3 $B_k = B_k^\star$. If two correct servers commit them in different views, then from Lemma 6 $B_k = B_k^\star$ holds. □

The liveness proof involves liveness for the view-leader as well as the round leaders.

Lemma 7 (Liveness for round leaders). *If L_r is a correct round leader, then L_r will always be able to propose blocks.*

Proof. The liveness property for round leaders follows trivially from Apollo in standard synchrony. □

Lemma 8 (Liveness for view leader). *If L_v is a correct view leader, then L_v will always be able to propose blocks.*

Proof. The proof follows trivially from Lemma 4.

Analogous to existing stable-leader based protocols [9,13,43], for theoretical liveness, the view-leaders must be changed regularly such as after every 1000 rounds, or a mechanism to blame the view leader needs to implemented [23] where the servers blame the view leader if a transaction from a client is not included within a certain time bound. □

Finally, we prove the liveness of Artemis. For completeness, we recall the liveness theorem here.

Theorem 5 (Artemis Liveness). *Assuming standard synchrony, Artemis always makes progress, and commits blocks with a period of at most $O(\kappa\Delta)$.*

Proof. The liveness proofs follow from Lemma 7 and Lemma 8. The $O(\kappa\Delta)$ time for the liveness parameter appears because in the worst case when κ consecutive view leaders are bad, there will be no valid blocks produced for $9(\kappa + 1)\Delta$ time. □