Moonshot: Optimizing Block Period and Commit Latency in Chain-Based Rotating Leader BFT

Isaac Doidge, Raghavendra Ramesh, Nibesh Shrestha and Joshua Tobkin
Supra Research
{i.doidge, r.ramesh, n.shrestha, j.tobkin}@supraoracles.com

***Abstract*—Existing chain-based rotating-leader BFT SMR protocols for the partially synchronous network model with constant commit latencies incur block periods of at least 2δ (where δ is the message transmission latency). While a protocol with a block period of δ exists under the synchronous model, its commit latency is linear in the size of the system.**

To close this gap, we present the first chain-based BFT SMR protocols with δ delay between the proposals of consecutive honest leaders and commit latencies of 3δ. We present three protocols for the partially synchronous model under different notions of optimistic responsiveness, two of which implement pipelining. All of our protocols achieve reorg resilience and two have short view lengths; properties that many existing chain-based BFT SMR protocols lack. We present an evaluation of our protocols in a wide-area network wherein they demonstrate significant increases in throughput and reductions in latency compared to the state-of-the-art, Jolteon. Our results also demonstrate that techniques commonly employed to reduce communication complexity—such as vote-pipelining and the use of designated vote-aggregators—actually reduce practical performance in many settings.

I. Introduction

Blockchain networks have become increasingly popular as mechanisms for facilitating decentralised, immutable and verifiable computation and storage. These networks leverage Byzantine fault-tolerant (BFT) consensus protocols to ensure that their participants (called *nodes*) execute the same sequence of operations (called *transactions*), despite some of them exhibiting arbitrary failures. Many blockchain networks also prioritize *fairness*; i.e., they strive to ensure that i) client transactions are processed promptly, without granting any client an unfair advantage over the others, and ii) nodes have an equal opportunity to be rewarded for the work that they do in the system. Public blockchain networks in particular also tend to be large, supporting hundreds (e.g. [27]) or thousands (e.g. [6]) of nodes in the pursuit of decentralization, and aim to cater to many concurrent clients. Accordingly, the consensus protocols driving these networks need to be efficient, maximising transaction throughput and minimising end-to-end commit latency (i.e., the time between a client submitting a transaction and it being executed by the blockchain).

To these ends, prior works [33], [38], [21], [4], [11], [22], [28] have leveraged two key strategies: i) *block chaining*, and; ii) frequent *leader rotation*. In the block chaining (or *chained*) paradigm, transactions are grouped into *blocks* that explicitly reference one or more existing blocks (called the *parents* of the block), typically by including their hashes. This enables an optimization called *pipelining*, wherein the *vote* acknowledgement messages sent by the nodes in the course of agreeing upon a block can be counted towards the finalization of its parents, facilitating the removal of additional voting phases and thus reducing the communication and computational complexity of the protocol by a constant factor. Our work focuses on the *chain-based* subcategory of chained protocols, wherein each block has exactly one parent, as opposed to DAG-based protocols in which a block may have many parents. In rotating-leader chain-based protocols, the leader responsible for proposing these blocks is changed at regular intervals, even when functioning correctly. This helps to fairly distribute the proposal workload and any related rewards. Additionally, the more frequently leaders are rotated the less amount of time a Byzantine (faulty) leader has to manipulate the ordering of pending transactions, improving censorship resistance. Accordingly, rotating-leader protocols often rotate the leader after every block proposal, an approach called *leader-speaks-once* (LSO). This paper seeks to optimize chain-based BFT consensus performance in a modified version of the LSO setting, which we name *leader-certifies-one* (LCO). Whereas an LSO protocol allows a leader to propose only a single block, an LCO protocol allows it to propose multiple but ensures that it produces no more than one certified block. Even as the previously cited works need not be implemented as LSO, our protocols need not be implemented as LCO, however, it is in this setting that they have the greatest advantage. We henceforth refer to chain-based BFT consensus protocols that implement leader rotation as *CRL protocols*.

Our work targets the *partially synchronous* network model [19] wherein there exists a time called the *Global Stabilization Time* (GST) after which message delivery takes at most Δ time. We use δ to denote the actual delivery time, which naturally satisfies $\delta \leq \Delta$ after GST. Many recent CRL protocols for this setting have focused on reducing communication complexity. Some have achieved linear communication complexity in their *steady state* phases [24], [21] (i.e. when the protocol makes progress under a fixed leader), while others obtain this result in their *view-change* phases [38], [28] (i.e. when the protocol elects a new leader) as well. However, these protocols sacrifice efficiency in several important metrics in their pursuit of linearity, including i) *minimum commit latency* (i.e., the minimum delay between a block being proposed and it being committed by all honest—i.e., non-faulty—nodes), ii) *minimum view change block period* (i.e., the minimum delay between the proposals of different honest leaders), and iii)

2158-3927/24/$31.00 ©2024 IEEE
DOI 10.1109/DSN58291.2024.00052

TABLE I
THEORETICAL COMPARISON OF CHAIN-BASED ROTATING LEADER BFT SMR PROTOCOLS

	Model	Minimum Commit Latency	Minimum View Change Block Period	Reorg Resilience	View Length	Pipelined	Communication Complexity$^{(1)}$		Optimistic Responsiveness	
							steady-state	view-change	standard	consecutive honest
HotStuff [38]	psync.	$7\delta^{(2)}$	2δ	✗	4Δ	✓	$O(n)$	$O(n)$	✓	✓
Fast HotStuff [24]	psync.	5δ	2δ	✗	4Δ	✓	$O(n)$	$O(n^2)$	✓	✓
Jolteon [21]	psync.	5δ	2δ	✗	4Δ	✓	$O(n)$	$O(n^2)$	✓	✓
HotStuff-2 [28]	psync.	5δ	2δ	✓	7Δ	✓	$O(n)$	$O(n)$	✗	✓
PaLa [14]	psync.	4δ	2δ	✓	5Δ	✓	$O(n^2)$	$O(n^2)$	✗	✓
ICC [11]	psync.	3δ	2δ	✗	4Δ	✗	$O(n^2)$	$O(n^2)$	✗	✓
Simplex [13]	psync.	3δ	2δ	✓	3Δ	✗	Unbounded$^{(3)}$	$O(n^2)$	✗$^{(4)}$	✓
Apollo [5]	sync.	$(f+1)\delta$	δ	✓	4Δ	✗	$O(n)$	$O(n^2)$	✗	✓
This work (§III)	psync.	3δ	δ	✓	5Δ	✓	$O(n^2)$	$O(n^2)$	✗	✓
This work (§IV)	psync.	3δ	δ	✓	3Δ	✓	$O(n^2)$	$O(n^2)$	✓	✓
This work (§V)	psync.	3δ	δ	✓	3Δ	✗	$O(n^2)$	$O(n^2)$	✓	✓

(1) Assuming the use of threshold signatures. (2) HotStuff has a minimum commit latency of 7δ if the next leader aggregates the votes for the current leader's proposal. In the original HotStuff specification, leaders aggregate the votes for their own proposals and then forward the resultant QC to the next leader, incurring an additional 3δ. (3) Simplex [13] requires each proposal to include its notarized parent blockchain, making the size of each proposal proportional to the size of the blockchain itself. (4) Simplex [13] claims responsiveness only when all nodes are honest.

view length (i.e., the duration a node waits in a view before it considers the current leader to have failed). In particular, these works require at least 5δ to commit a new block, at least 2δ between honest proposals in the LSO setting, and view lengths of at least 4Δ. Moreover, since these protocols all rely on a designated node to aggregate vote messages and forward the resulting certificates, they grant the adversary the power to censor certificates for honest proposals when this aggregator is Byzantine—even after GST. Accordingly, any implementation of these protocols that uses any node other than the original proposer as the vote aggregator is not *reorg resilient*; i.e., it cannot guarantee that an honest leader that proposes after GST will produce a block that becomes a part of the committed blockchain.

A recent line of work [11], [13] designed CRL protocols with minimum commit latencies of 3δ. However, these protocols are in the non-pipelined setting, have minimum view change block periods of 2δ and either have long view lengths [11] or are less practical in nature [13]. To the best of our knowledge, Apollo [5] is the only existing CRL protocol with a minimum view change block period of δ. However, it incurs a minimum commit latency of $(f+1)\delta$ even during failure-free executions and assumes a synchronous network. As far as we know, no chain-based consensus protocol has simultaneously achieved a minimum view change block period of δ and a constant commit latency. To close this gap, our paper explores the design of such protocols, which we collectively refer to as *Moonshot* protocols.

We first present two state machine replication (SMR) protocols for the pipelined setting, each of which satisfies a different notion of *responsiveness*: i) *optimistic responsiveness* [38] (Definition 6) and ii) *optimistic responsiveness under consecutive honest leaders* [22] (Definition 7). Informally, the former requires an honest leader to make progress in $O(\delta)$ time after GST (i.e., without waiting for $\Omega(\Delta)$ time) while the latter requires an honest leader to make progress in $O(\delta)$ time only when the previous leader is also honest. Our first protocol satisfies the former definition and is simpler to reason about, but has a longer view length. The second satisfies the latter definition and has a shorter view length, but is more complex.

Both of our protocols require only two consecutive honest leaders after GST to commit a new block, and achieve reorg resilience through vote-multicasting. This strategy, together with an optimization that we call *optimistic proposal*, also enables them to achieve both a minimum view change block period of δ and a minimum commit latency of 3δ. We say that a protocol implements optimistic proposal if a leader is allowed to "optimistically" extend a block proposed by its predecessor without waiting to observe its certification. We implement this in our protocols by allowing the leader of the next view to propose a new block when it votes for a block made by the leader of the current view.

As mentioned, pipelining can reduce the communication and computational overhead of a protocol. However, while this gives pipelined protocols good latency when all messages require a similar amount of time to propagate and process, pipelining actually increases commit latency when blocks take sufficiently longer to propagate or process than votes. Accordingly, we also present a non-pipelined variant of our second protocol in Section V. This final protocol retains standard optimistic responsiveness and requires only a single honest leader to commit a new block after GST.

Subsequently, we present an evaluation of LCO implementations of our protocols against an LSO implementation of Jolteon. Our protocols outperformed Jolteon in failure-free wide-area networks (WANs) of up to 200 nodes, committing approximately 1.5x as many blocks at around half the latency, on average. Our protocols also outperformed under failures, with our non-pipelined protocol committing 8x as many blocks with a reduction in latency by more than two orders of magnitude under Jolteon's worst-case leader schedule.

Organization. The rest of the paper is organized as follows: In Section II, we present the system model and preliminaries for our work. Section III presents a pipelined CRL protocol with a minimum commit latency of 3δ, minimum view change block period of δ, reorg resilience and optimistic responsiveness under consecutive honest leaders. We then modify this protocol in Section IV to obtain a protocol with standard optimistic responsiveness and improved view length. In Section V, we give a non-pipelined version of our second protocol, which offers improved commit latency when blocks take sufficiently longer to propagate or process than votes. We present an evaluation of our protocols in Section VI, and conclude with a more detailed discussion of related works in Section VII.

II. Preliminaries

We consider a system comprised of a set $\mathcal{V} = (P_1, \ldots, P_n)$ of n nodes running a protocol $\mathcal{P}$ in a reliable, authenticated all-to-all network. We assume the existence of a static, computationally bounded adversary that cannot break cryptographic primitives but may corrupt up to $f < n/3$ of the nodes when $\mathcal{P}$ begins, which it may then cause to behave arbitrarily. We refer to all nodes under the control of the adversary as being *Byzantine*, while we refer to those that adhere to $\mathcal{P}$ as being *honest*. We define a *quorum* as a set of $\lfloor \frac{n}{2} \rfloor + f + 1$ nodes. Henceforth, for the sake of simplicity, we assume that $n = 3f+1$ and that a quorum therefore contains $2f+1$ nodes.

We assume that each node has access to a local clock and that these clocks collectively have *no drift* and *arbitrary skew*. We also assume the *partially synchronous communication* model of Dwork et al. [19]. Under this model, the network starts in an initial state of asynchrony during which the adversary may arbitrarily delay messages sent by honest nodes. However, after an unknown time called the *Global Stabilization Time* (GST), the adversary must ensure that all messages exchanged between honest nodes are delivered within Δ time of being sent (from the perspective of the sender). In our initial analyses, we denote the range of the actual transmission latencies of messages of all types with δ, and observe that $\delta = [0, \Delta]$ after GST. Moreover, when we measure latency in terms of δ, e.g. $x = y\delta$, we are denoting that x requires the propagation of y sequential messages (i.e. x requires y network hops). In our later analyses we base our communication model on the *modified partially synchronous model* of Blum et al. [7]. Under this model, we denote the range of the actual delivery times of small messages (such as votes) with ρ and that of large messages (such as block proposals) with β, such that $\rho = [0, \min(\beta))$ and $\beta = (\max(\rho), \Delta]$, after GST. We follow a similar convention as with δ when measuring latency in terms of β and ρ, with $x = y\beta + z\rho$ denoting that x requires the sequential propagation of y large and z small messages.

We make use of digital signatures and a public-key infrastructure (PKI) to prevent spoofing and replay attacks and to validate messages. We use $\langle x \rangle_i$ to denote a message x digitally signed by node P_i using its private key. In addition, we use $\langle x \rangle$ to denote an unsigned message x sent via an authenticated channel. We use $H(x)$ to denote the invocation of the hash function H with input x.

A. Property Definitions

State Machine Replication. A state machine replication (SMR) protocol run by a network $\mathcal{V}$ of n nodes receives requests (transactions) from external parties, called *clients*, as input, and outputs a totally ordered log of these requests. We recall the definition of SMR given in [2], below.

Definition 1 (Byzantine Fault-Tolerant State Machine Replication [2])**.** *A Byzantine fault-tolerant state machine replication protocol commits client requests as a linearizable log to provide a consistent view of the log akin to a single non-faulty node, providing the following two guarantees.*

- ***Safety**. Honest nodes do not commit different values at the same log position.*
- ***Liveness**. Each client request is eventually committed by all honest nodes.*

Definition 2 (Minimum View Change Block Period (ω))**.** *The minimum view change block period ω of a chained consensus protocol $\mathcal{P}$ is the minimum latency between the proposal of a block B by an honest node P_i and its extension (directly or indirectly) by any honest node P_j such that $P_j \neq P_i$.*

Definition 3 (Minimum Commit Latency (λ))**.** *A consensus protocol has a minimum commit latency of λ if all honest nodes that commit a block proposed at time t, do so no earlier than $t + \lambda$.*

In this paper, we measure the above two metrics in relation to message transmission latency and assume that message processing time is relatively negligible.

Definition 4 (View Length (τ))**.** *A consensus protocol has a view length of τ if an honest node that enters view v at time t considers the view to have failed if it remains in v until $t + \tau$.*

Definition 5 (Reorg Resilience)**.** *We say that a consensus protocol is reorg resilient if it ensures that when an honest leader proposes after GST, one of its proposals becomes certified and this proposal is extended by every subsequently certified proposal.*

Optimistic Responsiveness. Responsiveness requires a consensus protocol to make progress in time proportional to the actual network delay (δ) and independent of any known upper bound delay (Δ) when a leader is honest [31]. Optimistic responsiveness requires this same guarantee, but only when certain optimistic conditions hold. Several variations [38], [4], [22], [13] have been formulated in the literature, two of which we make use of in this paper and recall below.

Definition 6 (Optimistic Responsiveness [38])**.** *After GST, any correct leader, once designated, needs to wait just for the first $n - f$ responses to guarantee that it can create a proposal that will make progress. This includes the case where a leader is replaced.*

We note that in [38], the term "make progress" means that all honest nodes will vote for the correct (honest) leader's proposal, not that all honest nodes observe a certificate for the included block; i.e. optimistic responsiveness does not imply reorg resilience. We also clarify that for LSO/LCO protocols, these (at most) $n-f$ responses should be messages from the previous view.

Definition 7 (Optimistic Responsiveness (Consecutive Honest) [4])**.** *We say that a protocol is optimistically responsive (consecutive honest) if after GST, for any two consecutive honest leaders L_v and L_{v+1}, L_{v+1} sends its proposal within $O(\delta)$ time of receiving L_v's proposal.*

Importantly, this variant of optimistic responsiveness allows the protocol to wait for $\Omega(\Delta)$ time before proposing in the new view when the leader of the previous view is Byzantine.

B. Protocol Definitions

View-based execution. Our protocols progress through a sequence of numbered *views*, with all nodes starting in view 1 and progressing to higher views as the protocol continues. Each view v is coordinated by a designated leader node L_v that is responsible for proposing a new block for addition to the blockchain. For the sake of liveness, we require that the leader election function L continually elects sequences of leaders that contain at least two consecutive (not necessarily distinct) honest leaders after GST for our pipelined protocols, and only one such leader for our non-pipelined protocol. We note that L must additionally change the leader every view for LCO implementations, and must elect each node with equal probability in fair implementations.

Blocks. The blockchains of each of our protocols are initialized with a *genesis block* B_0 that is known to all nodes at the beginning of the protocol. Each block references its immediate predecessor in the chain, which we refer to as its *parent*, with the parent of the genesis block being $\perp$. We say that a block *directly extends* its parent and *indirectly extends* its other predecessors in the chain. For simplicity when reasoning, we also say that a block extends itself. We refer to the predecessors of a given block as its *ancestors* and measure its *height* by counting its ancestors. A block B_k with height k has the format, $B_k := (b_v, H(B_{k-1}))$ where b_v is a fixed payload for the view v for which B_k is proposed, B_{k-1} is the parent of B_k, and $H(B_{k-1})$ is the hash digest of B_{k-1}. We allow the implementation to dictate the contents of b_v (e.g. transactions or hashes of batches of transactions). Accordingly, B_k is *valid* if i) its parent is valid, or if $k = 0$ and its parent is $\perp$, and ii) b_v satisfies the implementation-specific validity conditions. Finally, we say that two blocks B_k and $B'_{k'}$ proposed for the same view *equivocate* one another if they do not both have the same parent and payload.

Block certificates. In our protocols, a node sends a signed vote message to indicate its acceptance of a block. A block certificate $\mathcal{C}_v(B_k)$ for view v consists of a quorum of distinct signed vote messages for B_k for v. We use $\mathcal{C}_v$ to denote a block certificate for view v when knowledge of the related block is irrelevant to the context. We rank block certificates by their views such that $\mathcal{C}_v \leq \mathcal{C}_{v'}$ if $v \leq v'$. We provide more detailed definitions in the following sections where necessary.

Timeout messages and timeout certificates. Our protocols maintain the liveness SMR property by requiring nodes to request a new leader when they fail to observe progress in their current views after a certain amount of time. They do so by sending signed timeout messages for the view, the contents of which are protocol-specific. A view v timeout certificate, denoted $\mathcal{TC}_v$, consists of a quorum of distinct signed timeout messages for v, denoted $\mathcal{T}_v$.

III. Simple Moonshot

We now present Simple Moonshot (Figure 1), the first of our CRL protocols for the pipelined setting. Simple Moonshot achieves $\omega = \delta$, $\lambda = 3\delta$, reorg-resilience and responsiveness under consecutive honest leaders. We first discuss how our protocols obtain the former properties before elaborating on Simple Moonshot itself.

Towards achieving $\omega = \delta$ and $\lambda = 3\delta$. Prior CRL protocols require L_v to observe C_{v-1} before proposing during their happy paths (i.e. when views progress without any honest node sending a timeout message—as opposed to the *fallback path*). This is intended to help honest leaders create blocks that will become committed, but is unnecessarily strict for this purpose and naturally affects $\omega \geq 2\delta$ and $\lambda \geq 4\delta$ in the pipelined setting. Our protocols improve upon these results by requiring i) the leader of view v to propose a block for v, say B_k, upon voting for a block in $v-1$, say B_{k-1}, and; ii) nodes to multicast their votes. Allowing leaders to propose optimistically in this way enables voting for B_{k-1} to proceed in parallel with the proposal of B_k. Moreover, when the dissemination times of vote and proposal messages are equal (see Figure 2), having nodes multicast their votes ensures that if all honest nodes vote for B_{k-1} then they will all receive B_k at the time that they construct $\mathcal{C}_{v-1}(B_{k-1})$, allowing them to vote for B_k and L_{v+1} to propose immediately upon entering v. Hence, in the happy path, L_{v+1} proposes as soon as it receives L_v's proposal, giving our protocols an ω of δ. Furthermore, since our pipelined protocols require two consecutive views to produce certified blocks before a new block can be committed, requirements (i) and (ii) also give our protocols a λ of 3δ.

A. Protocol Details

We define Simple Moonshot in Figure 1 as a series of event handlers to be run by each node $P_i \in \mathcal{V}$. We elaborate below.

Advance View and Timeout. P_i enters view v from some view $v' < v$ upon receiving a view $v-1$ block certificate or a view $v-1$ timeout certificate (i.e. P_i never decreases its local view). Before doing so, it first multicasts this certificate. This ensures that if the first honest node enters v after GST then all honest nodes will enter v or higher within Δ thereafter, helping our protocol to obtain liveness and reorg resilience. Subsequently, P_i updates lock_i to the highest ranked block certificate that it has received so far and if lock_i is not $\mathcal{C}_{v-1}$ then P_i unicasts a status message containing lock_i to L_v. We note that P_i only updates lock_i during the view transition process and does not do so after entering the new view, even if it receives a higher ranked block certificate. This ensures that

A Simple Moonshot node P_i runs the following protocol whilst in view v:

1) **Propose.** If P_i is L_v and enters v at time t_i, propose: (i) upon receiving $\mathcal{C}_{v-1}(B_{k-1})$ before $t_i + 2\Delta$, or; (ii) at $t_i + 2\Delta$. Do so by multicasting $\langle \mathsf{propose}, B_k, \mathcal{C}_{v'}(B_{k-1}), v\rangle$, where $\mathcal{C}_{v'}(B_{k-1})$ is the highest ranked block certificate known to L_v and B_k extends B_{k-1}.
2) **Vote.** P_i votes once using one of the following rules:
   a) Upon receiving $\langle \mathsf{opt\text{-}propose}, B_k, v\rangle$ such that B_k extends B_{k-1}, if $\mathsf{lock}_i = \mathcal{C}_{v-1}(B_{k-1})$ then multicast $\langle \mathsf{vote}, H(B_k), v\rangle_i$.
   b) Upon receiving $\langle \mathsf{propose}, B_k, \mathcal{C}_{v'}(B_h), v\rangle$, if $\mathcal{C}_{v'}(B_h) \geq \mathsf{lock}_i$ and B_k extends B_h then multicast $\langle \mathsf{vote}, H(B_k), v\rangle_i$.
3) **Optimistic Propose.** Upon voting for B_k in v, if P_i is L_{v+1}, multicast $\langle \mathsf{opt\text{-}propose}, B_{k+1}, v+1\rangle$ such that B_{k+1} extends B_k.
4) **Timeout.** Upon receiving $f+1$ distinct $\langle \mathsf{timeout}, v\rangle_*$ or when $\mathsf{view\text{-}timer}_i$ expires, stop voting in v and multicast $\langle \mathsf{timeout}, v\rangle_i$.
5) **Advance View.** Upon receiving $\mathcal{C}_{v'-1}(B_h)$ or $\mathcal{TC}_{v'-1}$, where $v' > v$, and before executing any other rule, do the following: i) multicast the certificate; ii) update lock_i to the highest ranked block certificate received so far; iii) unicast a status message $\langle \mathsf{status}, v', \mathsf{lock}_i\rangle$ to $L_{v'}$ if lock_i has a view less than $v' - 1$, iv) enter v', and; v) reset $\mathsf{view\text{-}timer}_i$ to 5Δ and start counting down.

P_i additionally performs the following action in any view:

1) **Direct Commit.** Upon receiving $\mathcal{C}_{v-1}(B_{k-1})$ and $\mathcal{C}_v(B_k)$ such that B_k extends B_{k-1}, commit B_{k-1}.
2) **Indirect Commit.** Upon directly committing B_{k-1}, commit all of its uncommitted ancestors.

Fig. 1. The Simple Moonshot Protocol

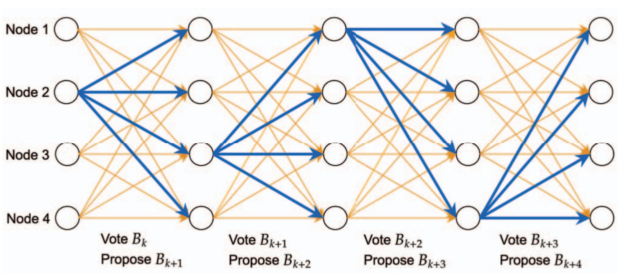


Fig. 2. Optimistic proposal (pictured in blue) and vote multicasting (pictured in orange) enable Simple Moonshot and Pipelined Moonshot to propose new blocks at the same rate that they become certified when proposals and votes take equal time to propagate and process.

the block certificate reported in a status message corresponds to its honest sender's lock_i for the duration of v, meaning that if L_v waits to receive status messages from all honest nodes before proposing, then it is guaranteed to extend the block certified by the highest ranked block certificate locked by any honest node. Finally, P_i enters v, resets $\mathsf{view\text{-}timer}_i$ to 5Δ and starts counting down. If L_v is honest and the network is synchronous then P_i should enter $v+1$ within 5Δ of entering v. If it does not, then it considers the current leader to have failed and so multicasts $\langle \mathsf{timeout}, v\rangle_i$ to request a view change and prevent the protocol from halting. P_i also does this whenever it observes that at least one other honest node has requested a view change for v.

Propose. Simple Moonshot allows two proposals to be created during view v: i) an optimistic proposal for view $v+1$, and; ii) a normal proposal for v. In the former case, L_{v+1} multicasts $\langle \mathsf{opt\text{-}propose}, B_{k+1}, v+1\rangle$, where B_{k+1} extends B_k, upon voting for B_k in v, hoping that B_k will become certified. When the protocol is operating in its happy path after GST, B_k will indeed become certified, enabling voting for consecutive honest proposals to proceed without delay. In the latter case, L_v multicasts $\langle \mathsf{propose}, B_h, \mathcal{C}_{v'}(B_{h-1}), v\rangle$, where B_h extends B_{h-1}, either upon receiving $\mathcal{C}_{v-1}(B_{h-1})$ within 2Δ time of entering v, or after having $\mathcal{C}_{v'}(B_{h-1})$ as its highest block certificate after waiting for 2Δ after entering v. Since messages are delivered within Δ time after GST, this 2Δ wait ensures that L_v will extend the highest certified block locked by any honest node when it proposes after GST, assisting with liveness and reorg resilience. We require L_v to multicast a normal proposal even when it has already multicasted an optimistic proposal to ensure that it always produces a certified block when it proposes after GST. We note that this requirement can be removed from each of our protocols to obtain the corresponding leader-speaks-once variant, but doing so naturally sacrifices reorg resilience because the adversary can cause optimistic proposals to fail, even after GST. We discuss how after introducing the remaining protocol rules.

Vote. Simple Moonshot has two rules for voting, at most one of which each node may invoke at most once per view. Firstly, P_i may vote for an optimistic proposal containing B_k proposed for view v and extending B_{k-1}, when locked on $\mathcal{C}_{v-1}(B_{k-1})$. In the best case, P_i receives the optimistic proposal containing B_k and $\mathcal{C}_{v-1}(B_{k-1})$ simultaneously and so votes for B_k immediately upon entering view v. Alternatively, if P_i receives $\langle \mathsf{propose}, B_h, \mathcal{C}_{v'}(B_{h-1}), v\rangle$ and $\mathcal{C}_{v'}(B_{h-1})$ ranks higher than or equal to lock_i and B_h extends B_{h-1}, then it votes for B_h. Importantly, if L_v creates both an optimistic proposal and a normal proposal with the same parent, then since payloads are fixed for a given view, both proposals will contain the same block. This ensures that all honest nodes will vote for the same block, even if they use different vote rules.

Commit. Finally, at any time during protocol execution, when an honest node P_i receives $\mathcal{C}_{v-1}(B_{k-1})$ and $\mathcal{C}_v(B_k)$, it commits B_{k-1} and all of its uncommitted ancestors. We say that a node *directly commits* B_k and *indirectly commits* any ancestors that it commits as a result of committing B_k.

B. Analysis

We now provide some discussion on the properties of Simple Moonshot, including brief intuitions for its safety, liveness and reorg resilience. We are unable to provide full proofs for our protocols in this paper due to space limitations, but the interested reader can find them in [18].

How does our protocol achieve safety? The vote and commit rules together ensure that Simple Moonshot satisfies the safety property of SMR. Specifically, if an honest node commits B_k for view v after receiving $\mathcal{C}_v(B_k)$ and $\mathcal{C}_{v+1}(B_{k+1})$, then a majority of the honest nodes must have voted for B_{k+1} in view $v+1$. Therefore, since honest nodes vote at most once per view, an equivocating $\mathcal{C}_v$ cannot exist so no honest node will be able to commit any block other than B_k for view v. Moreover, since the set of honest nodes that voted for B_{k+1}, say H, must have had $\mathcal{C}_v(B_k)$ when they voted for B_{k+1}, they will either lock this certificate or one of a higher rank upon transitioning from $v+1$ to a higher view. Once again then, since block certificates must contain votes from a majority of the honest nodes, every block certificate for every view greater than v must contain a vote from at least one member of H. Suppose that v' is the first view greater than $v+1$ to produce a block certificate and let P_i be a member of H that votes towards $\mathcal{C}_{v'}(B_l)$. Importantly, since P_i must lock $\mathcal{C}_v(B_k)$ before voting for B_l and since no higher ranked block certificate than $\mathcal{C}_{v+1}(B_{k+1})$ can exist before it does so, by the vote rules, B_l must directly extend either B_k or B_{k+1}. By extension then, every block certified for a higher view than v must extend B_k. This is sufficient to ensure safety.

Why propose twice? As previously mentioned, we require our leaders to make normal proposals even if they have already made an optimistic proposal because the adversary can cause optimistic proposals to fail even after GST. Suppose that an honest leader L_v proposes B_k extending B_{k-1} in an optimistic proposal. Per the optimistic vote rule, the adversary can cause B_k to fail by preventing some honest node from locking $\mathcal{C}_{v-1}(B_{k-1})$. This could happen either due to some other block, say B_l, becoming certified for view $v-1$, or due to the node entering v via $\mathcal{TC}_{v-1}$. In either case, since L_v is guaranteed to observe the highest ranked block certificate locked by any honest node upon entering view v, say $\mathcal{C}_{v'}(B_h)$, before it multicasts its normal proposal, it will be able to multicast a new block, say B_{h+1}, that extends B_h. Therefore, since honest nodes only update their locks when entering a new view, those that receive B_{h+1} whilst in view v will all have $\mathsf{lock}_i \leq \mathcal{C}_{v'}(B_h)$ and hence will vote for it. Thus, this requirement yields two important properties: i) that honest leaders are able to correct themselves when they initially extend a block that fails to become certified, and; ii) that if this block does become certified and its certificate is locked by any honest node, then the block included in the optimistic proposal will become certified even if some honest nodes initially fail to lock this certificate. This ensures that every honest leader that proposes after GST produces exactly one certified block.

How does our protocol achieve reorg resilience and liveness? As we have just explained, Simple Moonshot guarantees that every honest leader that proposes after GST produces exactly one certified block. Suppose that L_v is such an honest leader and produces $\mathcal{C}_v(B_k)$, and let t denote the time that the first honest node enters v. Since the multicasting of block certificates and timeout certificates ensures that all honest nodes will enter view v or higher within $t+\Delta$, if L_v is honest then it will send its last proposal by $t+3\Delta$, so all honest nodes will finish voting before $t+4\Delta$ and thus before any honest node can have sent $\mathcal{T}_v$ or higher. Therefore, either all honest nodes vote for B_k or some honest node must have entered $v+1$ via $\mathcal{C}_v(B_k)$ first. In either case, all honest nodes will receive $\mathcal{C}_v(B_k)$ within 5Δ of the first honest node entering v, so at least $f+1$ honest nodes will lock this certificate. Therefore, since these $f+1$ nodes will not vote for any optimistic proposal that does not directly extend their lock, and since no higher ranked block certificate can exist before they lock $\mathcal{C}_v(B_k)$, every certified block for every view greater than v must extend B_k satisfying Definition 5. Moreover, when L_{v+1} is also honest, it will necessarily be among the $f+1$ honest nodes that lock $\mathcal{C}_v(B_k)$ and will therefore multicast a proposal that extends B_k no later than the time that it enters $v+1$. Consequently, by the prior reasoning, all honest nodes will also receive $\mathcal{C}_{v+1}(B_{k+1})$ and thus will commit B_k. Accordingly, Simple Moonshot commits a new block whenever there are two consecutive honest leaders after GST, which is sufficient to ensure liveness.

IV. Pipelined Moonshot

Although Simple Moonshot has $\omega = \delta$, $\lambda = 3\delta$ and reorg resilience, it only provides responsiveness under consecutive honest leaders. If the leader of the current view fails then the next leader has to wait for $\Omega(\Delta)$ time to ensure that it can create a block that will become certified, naturally increasing τ. We now present Pipelined Moonshot (Figure 3), a CRL protocol that improves on Simple Moonshot in both of these areas to achieve full optimistic responsiveness and a τ of 3Δ.

Towards achieving optimistic responsiveness with $\tau = 3\Delta$. In Pipelined Moonshot, we separate the fallback case of Simple Moonshot's normal proposal into its own proposal type by enabling L_v to create a *fallback proposal* extending its lock upon entering v via $\mathcal{TC}_{v-1}$. While this means that L_v no longer needs to wait $\Omega(\Delta)$ time before proposing in the fallback path—making Pipelined Moonshot optimistically responsive per Definition 6—it also means that L_v may not receive the locks of all honest nodes before proposing. However, since $\mathcal{TC}$s must be constructed from $2f+1$ timeout messages, which in turn must now include the sender's lock, L_v must process the locks of at least $f+1$ honest nodes before creating its proposal. Consequently, L_v's lock is guaranteed to have a rank at least as great as the highest the highest ranked lock among these nodes at the time that they sent their timeout messages. This, along with the rules for voting, guarantees that there cannot exist a committable block with a higher height than L_v's lock. This helps to preserve safety in light of the additional modification that we make to preserve liveness, which we do by allowing P_i to vote for $\langle \mathsf{fb\text{-}propose}, B_k, \mathcal{C}_{v'}(B_h), \mathcal{TC}_{v-1}, v\rangle$ even if it has $\mathsf{lock}_i > \mathcal{C}_{v'}(B_h)$, given B_k directly extends B_h and $\mathcal{C}_{v'}(B_h)$ has a rank at least as great as the highest ranked block certificate included in $\mathcal{TC}_{v-1}$.

Requiring timeout messages to include block certificates naturally increases their size. Similarly, since $\mathcal{TC}$s must provably contain the highest ranked block certificate out of $2f+1$ timeout messages, they are necessarily linear in size even when

A Pipelined Moonshot node P_i runs the following protocol whilst in view v:

1) **Propose.** Upon entering v and after executing *Advance View* and *Lock*, if P_i is L_v, propose using one of the following rules:
   a) **Normal Propose.** If L_v entered v by receiving $\mathcal{C}_{v-1}(B_{k-1})$, multicast $\langle \mathsf{propose}, B_k, \mathcal{C}_{v-1}(B_{k-1}), v\rangle$ such that B_k extends B_{k-1}.
   b) **Fallback Propose.** If L_v entered v by receiving $\mathcal{TC}_{v-1}$, multicast $\langle \mathsf{fb\text{-}propose}, B_k, \mathcal{C}_{v'}(B_{k-1}), \mathcal{TC}_{v-1}, v\rangle$ such that $\mathcal{C}_{v'}(B_{k-1})$ is lock_i and B_k extends B_{k-1}.
2) **Vote.** P_i votes at most twice in view v when the following conditions are met:
   a) **Optimistic Vote.** Upon receiving $\langle \mathsf{opt\text{-}propose}, B_k, v\rangle$ such that B_k extends B_{k-1}, if (i) $\mathsf{timeout\_view}_i < v-1$, (ii) $\mathsf{lock}_i = \mathcal{C}_{v-1}(B_{k-1})$ and (iii) P_i has not voted in v, multicast $\langle \mathsf{opt\text{-}vote}, H(B_k), v\rangle_i$.
   b) After executing *Advance View* and *Lock* with all embedded certificates, vote once when one of the following conditions are satisfied:
      i) **Normal Vote.** Upon receiving $\langle \mathsf{propose}, B_k, \mathcal{C}_{v-1}(B_h), v\rangle$, if (i) $\mathsf{timeout\_view}_i < v$, (ii) B_k directly extends B_h and (iii) P_i has not sent an optimistic vote for an equivocating block $B'_{k'}$ in v, multicast $\langle \mathsf{vote}, H(B_k), v\rangle_i$.
      ii) **Fallback Vote.** Upon receiving $\langle \mathsf{fb\text{-}propose}, B_k, \mathcal{C}_{v'}(B_h), \mathcal{TC}_{v-1}, v\rangle$ if (i) $\mathsf{timeout\_view}_i < v$, (ii) B_k directly extends B_h and (iii) $\mathcal{C}_{v'}(B_h)$ has an equal or greater rank than the highest ranked certificate in $\mathcal{TC}_{v-1}$, multicast $\langle \mathsf{fb\text{-}vote}, H(B_k), v\rangle_i$.
3) **Optimistic Propose.** Upon voting for B_k in view v, if P_i is L_{v+1}, multicast $\langle \mathsf{opt\text{-}propose}, B_{k+1}, v+1\rangle$ such that B_{k+1} extends B_k.
4) **Timeout.** Upon the expiration of $\mathsf{view\text{-}timer}_i$, if P_i has not already sent $\mathcal{T}_v$, multicast $\langle \mathsf{timeout}, v, \mathsf{lock}_i\rangle_i$ and set $\mathsf{timeout\_view}_i = \max(\mathsf{timeout\_view}_i, v)$. Additionally, upon receiving $f+1$ distinct $\langle \mathsf{timeout}, v', \_\rangle_*$ messages or $\mathcal{TC}_{v'}$ such that $v' \geq v$ and not having sent $\mathcal{T}_{v'}$, multicast $\langle \mathsf{timeout}, v', \mathsf{lock}_i\rangle_i$ and set $\mathsf{timeout\_view}_i = \max(\mathsf{timeout\_view}_i, v')$.
5) **Advance View.** P_i enters v' where $v' > v$ using one of the following rules:
   - Upon receiving $\mathcal{C}_{v'-1}(B_h)$. Also, multicast $\mathcal{C}_{v'-1}(B_h)$.
   - Upon receiving $\mathcal{TC}_{v'-1}$. Also, unicast $\mathcal{TC}_{v'-1}$ to $L_{v'}$.

   Finally, reset $\mathsf{view\text{-}timer}_i$ to 3Δ and start counting down.

P_i additionally performs the following actions in any view:

1) **Lock.** Upon receiving $\mathcal{C}_v(B_k)$ in any protocol message whilst having $\mathsf{lock}_i = \mathcal{C}_{v'}(B_{k'})$ such that $v > v'$, set lock_i to $\mathcal{C}_v(B_k)$.
2) **Direct Commit.** Upon receiving $\mathcal{C}_{v-1}(B_{k-1})$ and $\mathcal{C}_v(B_k)$ such that B_k extends B_{k-1}, commit B_{k-1}.
3) **Indirect Commit.** Upon directly committing B_{k-1}, commit all of its uncommitted ancestors.

Fig. 3. The Pipelined Moonshot Protocol

using threshold signatures [8]. Accordingly, to avoid cubic communication complexity even under threshold signatures, our protocol replaces the $\mathcal{TC}$ multicast of Simple Moonshot with a Bracha-style amplification step [9]. In particular, P_i multicasts a $\mathcal{T}_v$ whilst in view v' where $v' \leq v$ when it first receives either $f+1$ $\mathcal{T}_v$ or $\mathcal{TC}_v$ from other nodes. This ensures that all honest nodes continue to enter new views after GST: In short, either all honest nodes will send view v Timeout messages, or, since we still require nodes to multicast block certificates, either some honest node must have observed and multicasted a view v or higher block certificate, or all honest nodes will send view v'' Timeout messages, where $v'' > v$.

A. Protocol Details

We now present the details of Pipelined Moonshot. We start with refinements to the definition of a block certificate and the certificate ranking rules before elaborating on the steps outlined in Figure 3 that differ from Simple Moonshot.

Block certificates. In Pipelined Moonshot, we use three types of signed vote messages: an optimistic vote ($\mathsf{opt\text{-}vote}$), a normal vote (vote) and a fallback vote ($\mathsf{fb\text{-}vote}$). Importantly, vote messages with different types may not be aggregated together. Accordingly, we now distinguish between three different types of block certificates. An *optimistic certificate* $\mathcal{C}^o_v(B_h)$ for a block B_h consists of $2f+1$ distinct $\mathsf{opt\text{-}vote}$ messages for B_h for view v. Similarly, a *normal certificate* $\mathcal{C}^n_v(B_h)$ consists of $2f+1$ distinct vote messages for B_h for view v. Finally, a *fallback certificate* $\mathcal{C}^f_v(B_h)$ consists of $2f+1$ distinct $\mathsf{fb\text{-}vote}$ messages for B_h for view v. We denote a block certificate with $\mathcal{C}_v(B_h)$ whenever its type is not relevant.

Locking. Simple Moonshot only allowed P_i to update lock_i upon entering a new view. In contrast, Pipelined Moonshot requires P_i to update lock_i upon receiving a higher ranked block certificate than its current lock_i, which may happen at any time during the protocol run.

Advance View and Timeout. As in Simple Moonshot, P_i enters view v from some view $v' < v$ upon receiving $\mathcal{C}_{v-1}$ or $\mathcal{TC}_{v-1}$. In the former case, as before, it then multicasts $\mathcal{C}_{v-1}$ to assist with reorg resilience and view synchronization. Comparatively, in the latter case P_i now unicasts $\mathcal{TC}_{v-1}$ to L_v instead of multicasting it. This helps to reduce the communication complexity of the protocol in light of its modified timeout messages, while still ensuring that L_v enters v within Δ of the first honest node doing so after GST. This in turn makes a $\mathsf{view\text{-}timer}$ of 3Δ sufficient to guarantee liveness, which P_i additionally resets regardless of how it enters v, and starts counting down. As before, if P_i does not advance to a new view before its view timer expires then it multicasts $\langle \mathsf{timeout}, v, \mathsf{lock}_i\rangle_i$. It likewise multicasts the same message for v'' upon observing evidence of at least one honest node requesting a view change for v'' such that $v'' \geq v$. This latter rule differs from Simple Moonshot and compensates for Pipelined Moonshot's removal of $\mathcal{TC}$ multicasting.

Propose. Pipelined Moonshot consists of three distinct ways to propose a new block in a view; i) an optimistic proposal, ii) a normal proposal, and iii) a fallback proposal. An honest node proposes using at most two of the three methods. The optimistic proposal rule remains the same as in Simple Moonshot and serves the same purpose, allowing voting to proceed without delay when network conditions are favourable. Comparatively, the normal proposal rule now only captures the first case of the same rule in Simple Moonshot: Namely, L_v multicasts a normal proposal $\langle \mathsf{propose}, B_k, \mathcal{C}_{v-1}(B_{k-1}), v\rangle$, where B_k extends B_{k-1}, upon entering view v via $\mathcal{C}_{v-1}(B_{k-1})$. As before, L_v does this even if it has already sent an optimistic proposal extending B_{k-1} (which, as before, will necessarily contain B_k). As in Simple Moonshot, this helps Pipelined Moonshot obtain reorg resilience by ensuring that, after GST, L_v will create a proposal that all honest nodes will vote for. Finally, L_v multicasts $\langle \mathsf{fb\text{-}propose}, B_h, \mathcal{C}_{v'}(B_{h-1}), \mathcal{TC}_{v-1}, v\rangle$, where B_h extends B_{h-1} and $\mathcal{C}_{v'}(B_{h-1})$ is lock_i, upon entering v via $\mathcal{TC}_{v-1}$. Importantly, since L_v only attempts this proposal after executing the *Lock* rule, $\mathcal{C}_{v'}(B_{h-1})$ is guaranteed to have a rank greater than or equal to that of the highest ranked certificate included in $\mathcal{TC}_{v-1}$.

Vote. In Pipelined Moonshot, P_i may vote up to twice in a view; at most once for an optimistic proposal and at most once for either a normal proposal or a fallback proposal. More precisely, P_i multicasts $\langle \mathsf{opt\text{-}vote}, H(B_k), v\rangle_i$ for $\langle \mathsf{opt\text{-}propose}, B_k, v\rangle$, where B_k extends B_{k-1}, when in view v if it has not yet sent a vote for v, or a timeout message for $v-1$ or higher, and has locked $\mathcal{C}_{v-1}(B_{k-1})$. As before, this enables P_i to vote for B_k immediately upon entering v in the best case. Additionally, P_i sends $\langle \mathsf{vote}, H(B_k), v\rangle_i$ for $\langle \mathsf{propose}, B_k, \mathcal{C}_{v-1}(B_{k-1}), v\rangle$ when in v if it has not sent either an $\mathsf{opt\text{-}vote}$ for an equivocating block in v or a timeout message for v or higher, and B_k extends B_{k-1}. Importantly, P_i must send this vote if it has already sent an optimistic vote for B_k. This ensures that B_k will be certified when L_v is honest and proposes after GST in the case where some honest nodes are unable to send an optimistic vote for B_k. Otherwise, P_i multicasts $\langle \mathsf{fb\text{-}vote}, H(B_h), v\rangle_i$ for $\langle \mathsf{fb\text{-}propose}, B_h, \mathcal{C}_{v'}(B_{h-1}), \mathcal{TC}_{v-1}, v\rangle$ when in v if it has not sent a timeout message for v or higher, B_h extends B_{h-1} and $\mathcal{C}_{v'}(B_{h-1})$ has a rank greater than or equal to that of the highest ranked block certificate in $\mathcal{TC}_{v-1}$. Notice that this rule allows P_i to send a fallback vote for B_h after having sent an optimistic vote for an equivocating block, say B_k. However, since the fallback proposal containing B_h can only be valid if it contains $\mathcal{TC}_{v-1}$, at least $f+1$ honest nodes must have sent $\mathcal{T}_{v-1}$ before entering v and thus will not be able to trigger the optimistic vote rule for B_k, so $\mathcal{C}^o_v(B_k)$ will never exist.

B. Analysis

Why is it safe to vote for a fallback proposal? As we mentioned earlier, we require honest nodes to vote for valid fallback proposals even when they are locked on a higher ranked block certificate than that of the parent of the proposed block. This remains safe because a fallback proposal must be justified by a $\mathcal{TC}$ for the previous view, which in turn contains information about the locks of a majority of the honest nodes. Specifically, $\mathcal{TC}_v$ guarantees that at least $f+1$ nodes had yet to vote for a higher height than $h+1$ upon sending $\mathcal{T}_v$, where h is the height of $\mathcal{C}_{v'}(B_h)$, the highest ranked block certificate included in $\mathcal{TC}_v$. Consequently, there cannot exist a committable block for any height greater than h when $\mathcal{TC}_v$ is constructed. Moreover, if any block can be committed at height h then there can be only one such block. This is because the commit rule only allows a block at height h proposed for view v'' to be committed if its child becomes certified in $v''+1$. Therefore, if B_h can be committed then at least $f+1$ honest nodes must have voted for its child in $v'+1$, and since an honest node cannot vote for a block unless it possesses the block certificate for its parent, these nodes must have had $\mathcal{C}_{v'}(B_h)$ when they did so. Consequently, every $\mathcal{TC}$ for $v'+1$ or higher will necessarily contain $\mathcal{C}_{v'}(B_h)$ or a block certificate for one of its descendants as its highest ranked block certificate, meaning that every fallback proposal for $v'+1$ or higher will necessarily extend B_h. Moreover, by extension, so will every subsequent optimistic or normal proposal.

Commit Moonshot can be obtained by adding the following rules to the protocol for P_i presented in Figure 3:

1) **Direct Pre-commit.** Upon receiving $\mathcal{C}_v(B_k)$ whilst in any view v' such that $v' \leq v$, if $\mathsf{timeout\_view}_i < v$, multicast $\langle \mathsf{commit}, H(B_k), v\rangle_i$.
2) **Indirect Pre-commit.** Upon receiving $\mathcal{C}_v(B_k)$ whilst in any view, having multicasted a commit vote for any descendant of B_k, having $\mathsf{timeout\_view}_i < v$ and having not yet multicasted $\langle \mathsf{commit}, H(B_k), v\rangle_i$, multicast $\langle \mathsf{commit}, H(B_k), v\rangle_i$.
3) **Alternative Direct Commit.** Upon receiving a quorum of distinct $\langle \mathsf{commit}, H(B_k), v\rangle_*$ whilst in any view, commit B_k.

Fig. 4. Commit Moonshot

V. Commit Moonshot

Until now, we have measured λ in terms of δ. However, this is imprecise because δ provides no way of differentiating between the performance of protocols that exchange one type of message for another. The pipelining technique fundamentally facilitates the removal of one or more phases from a protocol by granting another phase additional meaning. In existing pipelined consensus protocols, this technique replaces two (or more) consecutive phases of voting for one block proposal, with one phase of voting for two (or more) consecutive block proposals. This means that the commit latency of a block in the pipelined setting is proportional to the dissemination time of not only the block itself, but also its child (in the best case). More to the point, pipelining essentially exchanges the cost of disseminating additional votes for the cost of disseminating additional proposals and thus naturally increases commit latency when proposals take sufficiently longer to disseminate than votes.

We characterize this behavior using a communication model based on the *modified partially synchronous model* [7] in which we assume that small messages (here, votes) are delivered within ρ time while large messages (here, block proposals) are delivered within β time such that $\rho = [0, \min(\beta))$ and $\beta = (\max(\rho), \Delta]$, after GST. Under this model, Simple Moonshot and Pipelined Moonshot both incur $\lambda = 2\beta + \rho$.

We now present a protocol with $\lambda = \beta + 2\rho$, which we call Commit Moonshot. Accordingly, when $\rho < \beta$ (as in Figure 5), which we assume is typical in practice, this protocol provides

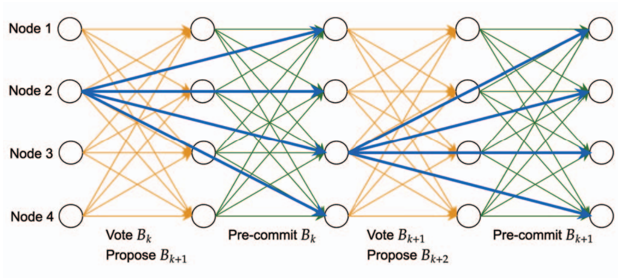


Fig. 5. Explicit commit votes (pictured in green) enable Commit Moonshot to commit blocks sooner than its pipelined counterparts when block proposals (pictured in blue) take sufficiently longer to disseminate than votes.

improved commit latency over those previously presented by integrating an explicit *pre-commit* phase. Like its counterparts, Commit Moonshot also obtains $\omega = \delta$ (β) and provides both reorg resilience and optimistic responsiveness. Additionally, while Simple Moonshot and Pipelined Moonshot require two consecutive honest leaders to guarantee a commit after GST, Commit Moonshot requires only one.

We present the modifications required to convert Pipelined Moonshot to Commit Moonshot in Figure 4. Since Commit Moonshot retains the rules of Pipelined Moonshot, the same liveness argument that can be made for the latter also applies to the former. However, the introduction of a secondary commit path demands additional reasoning about the safety of the protocol. We present a brief intuition to this end below, and provide full reasoning in [18].

Safety intuition. Per the alternative commit rule given in Figure 4, P_i commits B_k and all of its uncommitted ancestors upon receiving a quorum (i.e. $2f+1$ when $n = 3f+1$) of distinct $\langle \mathsf{commit}, H(B_k), v \rangle_*$ messages. This remains safe because $2f+1$ such messages can only exist if at least $f+1$ honest nodes do not send $\mathcal{T}_v$. Consequently, if any block becomes certified for $v+1$ then it must have been proposed in either an optimistic or normal proposal and thus must be a child of B_k. Otherwise, $\mathcal{TC}_{v+1}$ will contain $\mathcal{C}_v(B_k)$ as its highest ranked block certificate and therefore every subsequently certified block will necessarily extend B_k.

VI. Implementation and Evaluation

As shown in Table I, Pipelined Moonshot and Commit Moonshot equal or surpass the theoretical performance of prior $O(n^2)$ CRL protocols in all considered metrics. The primary question that remains, then, is whether their increased communication complexity relative to linear protocols is justified. Accordingly, we decided to implement our protocols and evaluate them against Jolteon, a linear protocol with state-of-the-art performance in most metrics and several high-quality open-source implementations.

Implementation. We implemented all three of our protocols by modifying the code for Jolteon available in the Narwhal-HotStuff branch of the repository [32] created by Facebook Research for evaluating Narhwal and Tusk [16]. We decoupled our implementation from Narhwal and did the same for Jolteon

TABLE II
Observed Latencies (in ms) between AWS regions

	Destination*				
Source	us-e-1	us-w-1	eu-n-1	ap-ne-1	ap-se-2
us-east-1	5.23	61.87	113.78	167.6	197.42
us-west-1	62.88	3.69	172.17	109.89	141.54
eu-north-1	114.09	173.31	5.48	248.67	271.68
ap-northeast-1	168.04	109.94	251.63	5.99	111.67
ap-southeast-2	199.54	146.06	272.31	112.11	4.53

*Region names are abbreviated versions of the Source regions.

so that we could compare the two consensus protocols in isolation. We replaced both the Narwhal mempool and the simulated-client process by having the leaders of each protocol create parametrically sized payloads during the block creation process, with individual payload items being 180 bytes in size. We used ED25519 signatures and constructed certificate proofs from an array of these signatures. We left the TCP-based network stack mostly intact and applied the few necessary changes to both implementations to ensure that any differences in performance were solely due to the differences between the consensus protocols themselves.

Setting. We chose to perform our evaluation in a setting typical of modern low-latency public blockchains such as Aptos [20] to demonstrate the efficacy of our protocols when network latency is the dominating performance factor. Accordingly, we constructed moderately-sized (up to 200 nodes) wide-area networks of nodes with high bandwidth capabilities and moderate computational capabilities. Specifically, we distributed the nodes evenly across the us-east-1 (N. Virginia), us-west-1 (N. California), eu-north-1 (Stockholm), ap-northeast-1 (Tokyo) and ap-southeast-2 (Sydney) AWS regions, with each node being allocated its own m5.large EC2 instance and connected to every other node via a separate point-to-point link. Each instance ran Ubuntu 20.04 and had a network bandwidth of up to 10Gbps[1], 8GB of memory and Intel Xeon Platinum 8000 series processors with 2 virtual cores. Table II reports the typical (90th percentile) latencies observed between these regions around the time of our experiments.

Variables and metrics. We first evaluated the trade-off between λ, ω and steady-state communication complexity in this setting by running all protocols with $f' = 0$, where f' denotes the number of actual failures in the system (i.e. $f' \leq f = \lfloor \frac{n-1}{3} \rfloor$), under varying network and payload sizes. Subsequently, we evaluated the impact of τ, reorg resilience, pipelining and optimistic responsiveness by running all protocols in a fixed network with $f' = f$ and varying leader schedules. We measured these trade-offs by comparing the throughput and latency of each protocol and established two metrics for throughput: Firstly, the number of blocks committed by at least $2f+1$ nodes during a run, hereafter referred to as *throughput*; and secondly, the average number of bytes of payload data from (subsequently) committed blocks

[1]https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-network-bandwidth.html

TABLE III
PERFORMANCE VS JOLTEON ($f' = 0$, OUTLIERS REMOVED)

	Throughput Increase (%)				Latency Reduction (%)			
Prot.	**Max**	$\bar{x}$	$\tilde{x}$	**Min**	**Max**	$\bar{x}$	$\tilde{x}$	**Min**
SM	72	53	55	33	56	43	42	37
PM	70	51	54	24	56	43	42	32
CM	74	52	54	25	69	54	58	38

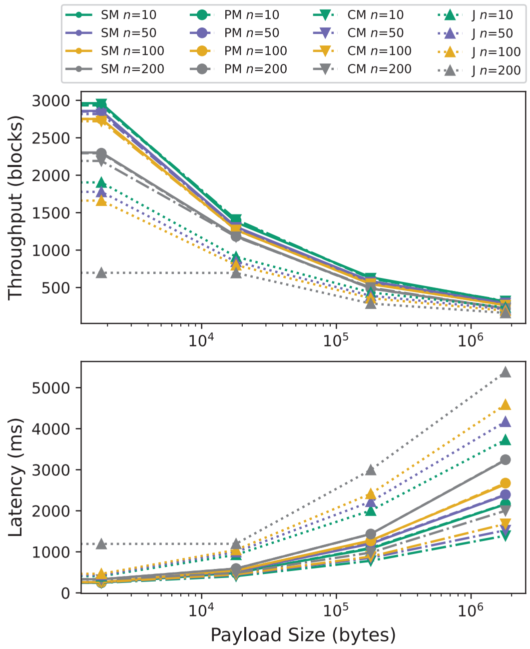


Fig. 6. Performance Overview ($f' = 0$, $p \leq 1.8$MB). Key trends: (1) Throughput approximately halves and latency roughly doubles for every order of magnitude increase in p. (2) Performance in both metrics decreases for all protocols as the network size increases. (3) Our protocols perform similarly in terms of throughput; Commit Moonshot achieves increasingly better latency as p increases. (4) Our protocols outperform Jolteon in both metrics.

transferred per second (i.e., throughput $\times$ payload size $\div$ runtime), hereafter referred to as *transfer rate*. For latency, we measured the average time between the creation of a block and its commit by the $(2f+1)$-th node. The plotted results are the averages of the related metrics across three five minute runs for each related configuration of the system.

We refer to Simple Moonshot, Pipelined Moonshot, Commit Moonshot and Jolteon as SM, PM, CM and J in the accompanying figures and tables.

A. Happy Path Evaluation ($f' = 0$)

We initially tested networks of 10, 50, 100 and 200 honest nodes with block payload sizes ranging from empty to 1.8MB to understand how the tested protocols scale under an increasing communication load. Figure 6 reports the results of these experiments. We subsequently tested additional payload sizes in the 200 node network to discover the approximate maximum transfer rate of each protocol in this setting, which can be seen in Figure 8.

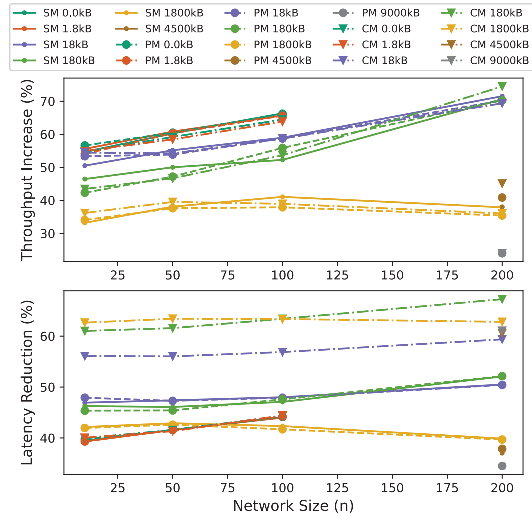


Fig. 7. Performance vs. Jolteon ($f' = 0$, No Outliers)

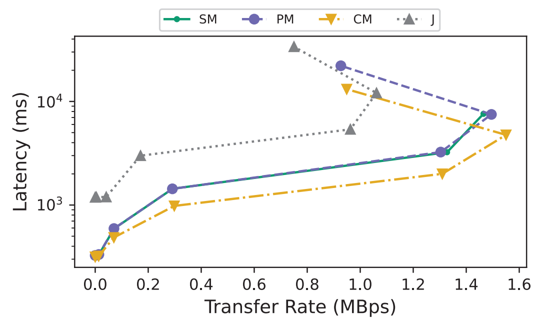


Fig. 8. Throughput vs Latency ($n = 200$, $f' = 0$, $p \leq 9$MB)

As shown in Figure 6, Figure 7 and Table III, all Moonshot protocols produced notably higher throughput than Jolteon in all tested configurations due to the more frequent block production afforded by their reduced ω. Likewise, the reduction in λ achieved by multicasting votes (in conjunction with optimistic proposals, in the case of the pipelined protocols) also caused them to produce substantially decreased latency compared to Jolteon across all configurations. The 200 node network produced significant outliers under the empty and 1.8kB payload configurations, with all three protocols exhibiting about thrice the throughput and a quarter of the latency of Jolteon, compared to the approximately 50% *increase in throughput* and $40\% - 50\%$ *reduction in latency* seen on average across all other configurations. Simple Moonshot and Pipelined Moonshot produced near-identical performance in both metrics for most configurations due to the similarity of their happy-path protocols. Conversely, although Commit Moonshot produced similar throughput to these protocols, it exhibited substantially reduced latency for payloads above 18kB due to its explicit commit messages, clearly showing the inefficiency of pipelining when blocks are large. Generally speaking, all three Moonshot protocols produced increasingly higher throughput and relatively consistent improvements to latency compared to Jolteon as the network size increased, showing that obtaining linear communication complexity is counter-productive in WANs of this scale if it comes at the cost of sequentializing network operations (i.e., reducing ω and λ). Finally, per Figure 8, all three Moonshot protocols achieved a *higher maximum transfer rate with lower latency* than Jolteon in the 200 node network, with Commit Moonshot producing the best results. Overall, these results show that the happy paths of our Moonshot protocols scale well and provide meaningfully decreased latency and increased throughput compared to Jolteon under the experimental conditions, with Commit Moonshot being the most efficient option.

B. Evaluation Under Failures ($f' = f$)

We subsequently further evaluated the impact of pipelining along with τ, reorg resilience and optimistic responsiveness, by running all protocols with a fixed n, f', p (i.e., block payload size) and Δ under three different fair LSO/LCO leader schedules. We chose $n = 100$, $f' = 33$ and $p = 0$ to maximize the impact of the quadratic steady-state complexity of our protocols without risking a repeat of the outliers seen in the $n = 200$, $f' = 0$ experiments. We also chose $\Delta = 500ms$, a somewhat-conservative value (per Table II) that still ensured that each protocol would make it through several iterations of the leader schedules within the five minute duration of each run. As for the leader schedules, the first ($\mathcal{B}$) had all honest nodes followed by all byzantine nodes, representing the best case for non-reorg-resilient and pipelined protocols. The second ($\mathcal{WM}$) had honest-then-byzantine leaders for $2f'$ views, followed by honest leaders for the remaining $n - 2f'$ views, representing the worst case for reorg resilient, pipelined protocols. The third ($\mathcal{WJ}$) repeated two-honest-then-byzantine for $3f'$ views, followed by the remaining $n - 3f'$ honest, representing the worst case for non-reorg resilient, pipelined protocols.

As shown in Figures 9a and 9b, Jolteon's performance degrades enormously in the presence of failures due to its lack of reorg resilience. This is evident by the difference in its results for $\mathcal{B}$ and $\mathcal{WJ}$, with the former producing approximately seven times higher throughput and fifty times lower latency than the latter. The pipelined nature of Simple Moonshot and Pipelined Moonshot likewise caused a significant reduction in latency between the worst ($\mathcal{WM}$) and best case ($\mathcal{B}$) leader schedules for these protocols. Simple Moonshot's 2Δ wait after a failed leader (i.e. lack of Optimistic Responsiveness) caused its performance to vary more significantly than Pipelined Moonshot, while its longer view length caused a substantial decrease in throughput.

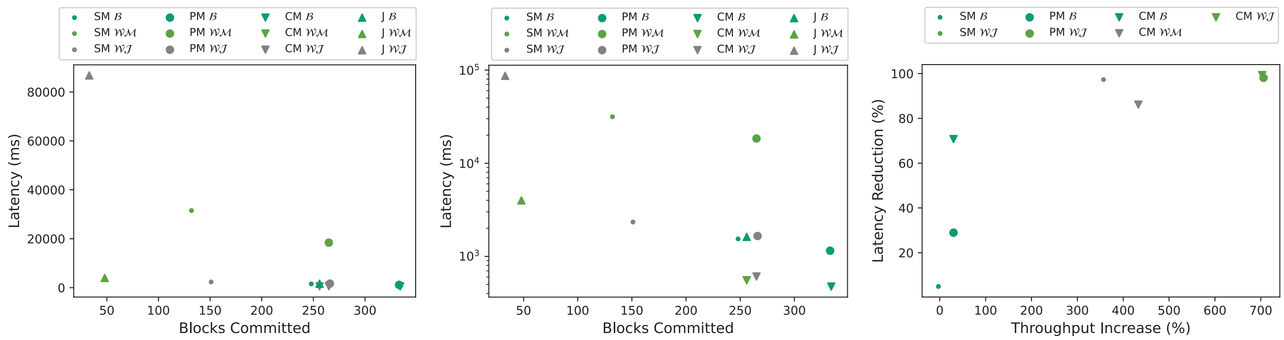
As shown by their absence from Figure 9c, both Simple Moonshot and Pipelined Moonshot failed to improve over Jolteon under $\mathcal{WM}$. More precisely, although they both produced a several-fold increase in throughput compared to Jolteon, Jolteon produced much lower latency. Both of these results were a side-effect of reorg resilience: Both Moonshot protocols committed all blocks proposed by honest leaders with Byzantine successors under this schedule, but only after a significant delay. Comparatively, Jolteon lost all such blocks due to lacking this property, with only the block of the final honest leader in the schedule being committed with a delay. Since Jolteon commits $n - 2f'$ out of every n blocks under this schedule, its relative improvement in block commit latency should increase proportionally to n, while its relative throughput should similarly decrease. However, we note that in this case reduced block commit latency at the cost of decreased throughput should be considered an undesirable trade-off as it does not imply a reduction in transaction commit latency.

Finally, Commit Moonshot performed consistently well regardless of the leader schedule due to its explicit pre-commit phase, which denies the adversary any power to delay the commit of honest blocks. Notably, as shown in Figure 9c, it produced around *eight times higher throughput* and more than *two orders of magnitude lower latency* than Jolteon under $\mathcal{WJ}$. Overall, then, Commit Moonshot produced superior performance in both the happy path and in the presence of failures, making it a prime candidate for application in modern low-latency public blockchains.

VII. Related Work

There has been a long line of work towards designing efficient BFT SMR protocols for partially synchronous networks (which we cite further on). Our work contributes to this effort by introducing the first CRL protocols to obtain both $\omega = \delta$ and $\lambda = 3\delta$. Our protocols further provide reorg resilience, improving their recovery time after a failed leader compared to prior chain-based works that fail to achieve this property. This is especially true of both Pipelined Moonshot and Commit Moonshot, which also have low τ and are optimistically responsive. These properties come at the cost of $O(n^2)$ steady-state communication complexity, making our protocols less performant in this metric compared to vote-aggregator-based protocols like HotStuff. However, as shown in Section VI, this trade-off is worthwhile in many settings. We presented a brief comparison between our protocols and other recent works in Section I. We now undertake a more thorough review.

Early works. PBFT [12] was the first practical BFT SMR protocol, achieving $\lambda = 3\delta$ at the cost of $O(n^2)$ steady-state communication. PBFT's slot-based nature complicated its view change, leading it to only rotate leaders after a failure—an approach that allows proposal frequency to be reduced below δ, but precludes fairness. Much later, Tendermint [10] combined the steady-state and view-change sub-protocols into a unified protocol for the LSO setting, resulting in a simpler protocol than PBFT at the cost of an $\Omega(\Delta)$ wait before every new view at the same height, thus sacrificing optimistic responsiveness. HotStuff [38] formalized the notion of optimistic responsiveness and improved upon Tendermint both by implementing this property and being the first protocol to obtain linear ($O(n)$) communication complexity in both its steady-state and view-change phases (in the presence of an abstract



(a) Performance Overview

(b) Performance Overview (log scaled)

(c) Improvements vs. Jolteon

Fig. 9. Performance comparison at $n = 100$, $f' = 33$ and $p = 0$

pacemaker for view synchronization). To our knowledge, it was also the first protocol to implement block chaining.

Linear protocols. Like HotStuff, many other chain-based protocols [23], [24], [21], [36], [28] have focused on minimising communication complexity, with some achieving linearity only in their steady states and others during their view-change phases as well. Recently, some [15], [28] have even achieved amortized-linear view synchronization. In all cases though, these protocols obtain steady-state linearity through the use of a designated vote-aggregator node. As we previously observed, this naturally increases their λ, ω and τ relative to our protocols, and precludes reorg resilience when the aggregator is not the original proposer. Moreover, while most nodes incur a steady-state complexity of $O(1)$ in these protocols, the proposer must still send and the aggregator must still receive, $O(n)$ messages. This imbalance means that these protocols under-utilize the available bandwidth in the point-to-point CRL setting (in which there should be no choke-points in the network and each node should have similar capabilities).

Non-linear pipelined chain-based protocols. PaLa [14] is a pipelined CRL protocol with $\lambda = 4\delta$ and $\omega = 2\delta$. While this improves upon the commit latencies of linear pipelined protocols with $\omega = 2\delta$, like [21], PaLa achieves this result at the cost of $O(n^2)$ communication complexity in its steady state. Accordingly, PaLa is sub-optimal in all three properties.

Non-pipelined chain-based protocols. Similar to PaLa, ICC [11] incurs $O(n^2)$ steady-state communication complexity. However, this protocol eschews pipelining, allowing it to achieve $\lambda = 3\delta$ through the use of an explicit second round of voting for each block. Even so, it lacks reorg resilience and its ω of 2δ and τ of 4Δ make it less efficient in these metrics than our protocols. Simplex [13] obtains the same λ and ω with $\tau = 3\Delta$, however, it claims responsiveness only when all nodes are honest. Additionally, its requirement that a leader must send the entire certified blockchain along with its proposal makes its communication complexity proportional to size of the blockchain and thus unbounded, rendering it impractical.

Apollo [5]. Apollo obtains $\omega = \delta$ at the cost of a $\lambda = (f+1)\delta$ and assuming a synchronous communication model.

DAG-based protocols. DAG-based consensus protocols like [34], [35], [29], [25] focus on improving block throughput. While they naturally produce and commit more blocks over a given interval than chain-based protocols by virtue of having all nodes propose in each step, they incur $O(n^3)$ communication in doing so. While recent protocols [25], [29] in this setting have achieved $\omega = \delta$, and $\lambda = 3\delta$ for blocks proposed by the leader, they require at least 4δ to commit blocks proposed by other nodes. Consequently, since most blocks committed by these protocols are non-leader blocks, their average block commit latency is still higher than our protocols. Moreover, since these protocols use pipelining, each δ corresponds to one β under our model from Section V, meaning that these latencies become even more significant relative to our protocols as blocks become larger.

Inspiration for future work. Moonshot may be further optimized by applying insights from other works. For example, a related line of works [1], [26], [30], [3], [23] achieve $\lambda = 2\delta$ via optimistic commits when $n \geq 5f - 1$. Similarly, works such as [37], [17] have leveraged trusted execution environments to limit the power of the adversary, enabling consensus when $n \geq 2f + 1$. Giridharan et al. also recently proposed BeeGees [22], a pipelined CRL protocol that is able to commit without requiring consecutive honest leaders. Defining variants of Moonshot that leverage these optimizations represents an interesting direction for future work.

VIII. Conclusion

We presented the first chain-based rotating leader BFT protocols for the partially synchronous network model with $\omega = \delta$ and $\lambda = 3\delta$. All three of our protocols outperformed the previous state-of-the-art CRL protocol, Jolteon, both in the presence of failures and in failure-free scenarios in a WAN setting. Pipelined Moonshot consistently outperformed Jolteon, showing the value of low ω and reorg resilience. Likewise, Commit Moonshot equalled or outperformed Pipelined Moonshot in all experiments, showing that pipelining is counter-productive in the presence of failures and when blocks are large. These results show that our protocols are suitable for application in modern low-latency public blockchain systems.

References

[1] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. *ACM SIGOPS Operating Systems Review*, 39(5):59–74, 2005.

[2] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In *IEEE S&P*, pages 106–118. IEEE, 2020.

[3] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of byzantine broadcast: A complete categorization. In *PODC*, pages 331–341, 2021.

[4] Ittai Abraham, Kartik Nayak, and Nibesh Shrestha. Optimal good-case latency for rotating leader synchronous bft. In *OPODIS*, 2022.

[5] Adithya Bhat, Akhil Bandarupalli, Saurabh Bagchi, Aniket Kate, and Michael Reiter. Unique chain rule and its applications. In *FC*, 2023.

[6] bitfly gmbh. Validators chart. https://beaconcha.in/charts/validators. [Online; accessed 04-December-2023].

[7] Erica Blum, Derek Leung, Julian Loss, Jonathan Katz, and Tal Rabin. Analyzing the real-world security of the algorand blockchain. In *CCS*, pages 830–844, 2023.

[8] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *Journal of cryptology*, 17:297–319, 2004.

[9] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[10] Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, University of Guelph, 2016.

[11] Jan Camenisch, Manu Drijvers, Timo Hanke, Yvonne-Anne Pignolet, Victor Shoup, and Dominic Williams. Internet computer consensus. In *PODC*, pages 81–91, 2022.

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

[13] Benjamin Y Chan and Rafael Pass. Simplex consensus: A simple and fast consensus protocol. In *TCC 2023 (To appear)*. Springer, 2023.

[14] TH Hubert Chan, Rafael Pass, and Elaine Shi. Pala: A simple partially synchronous blockchain. *Cryptology ePrint Archive*, 2018.

[15] Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. Byzantine consensus is θ (n^2): The dolev-reischuk bound is tight even in partial synchrony! In *DISC*, 2022.

[16] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Eurosys*, pages 34–50, 2022.

[17] Jérémie Decouchant, David Kozhaya, Vincent Rahli, and Jiangshan Yu. Damysus: streamlined bft consensus leveraging trusted components. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 1–16, New York, NY, USA, 2022. Association for Computing Machinery.

[18] Isaac Doidge, Raghavendra Ramesh, Nibesh Shrestha, and Joshua Tobkin. Moonshot: Optimizing chain-based rotating leader bft via optimistic proposals. *arXiv preprint arXiv:2401.01791*, 2024.

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

[20] Aptos Foundation. Aptos: The world's most production-ready blockchain. https://aptosfoundation.org/. [Online; accessed 28-February-2024].

[21] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *FC*, pages 296–315, 2022.

[22] Neil Giridharan, Florian Suri-Payer, Matthew Ding, Heidi Howard, Ittai Abraham, and Natacha Crooks. Beegees: stayin'alive in chained bft. In *PODC*, pages 233–243, 2023.

[23] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: A scalable and decentralized trust infrastructure. In *DSN*, pages 568–580. IEEE, 2019.

[24] Mohammad M Jalalzai, Jianyu Niu, Chen Feng, and Fangyu Gai. Fast-hotstuff: A fast and resilient hotstuff protocol. *arXiv preprint arXiv:2010.11454*, 2020.

[25] Idit Keidar, Oded Naor, Ouri Poupko, and Ehud Shapiro. Cordial miners: Fast and efficient consensus for every eventuality. In *DISC*, 2023.

[26] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *SOSP*, pages 45–58, 2007.

[27] Aptos Labs. Validators. https://explorer.aptoslabs.com/validators/all?network=mainnet. [Online; accessed 04-December-2023].

[28] Dahlia Malkhi and Kartik Nayak. Hotstuff-2: Optimal two-phase responsive bft. *Cryptology ePrint Archive*, 2023.

[29] Dahlia Malkhi, Chrysoula Stathakopoulou, and Maofan Yin. Bbca-chain: One-message, low latency bft consensus on a dag. *arXiv preprint arXiv:2310.06335*, 2023.

[30] J-P Martin and Lorenzo Alvisi. Fast byzantine consensus. *TDSC*, 3(3):202–215, 2006.

[31] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. In *DISC*, page 6, 2017.

[32] Facebook Research. Narwhal-hotstuff github repository. https://github.com/facebookresearch/narwhal/tree/narwhal-hs. [Online; accessed 22-January-2023].

[33] Elaine Shi. Streamlined blockchains: A simple and elegant approach (a tutorial and survey). In *ASIACRYPT*, pages 3–17. Springer, 2019.

[34] Alexander Spiegelman, Balaji Aurn, Rati Gelashvili, and Zekun Li. Shoal: Improving dag-bft latency and robustness. *arXiv preprint arXiv:2306.03058*, 2023.

[35] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *CCS*, pages 2705–2718, 2022.

[36] Xiao Sui, Sisi Duan, and Haibin Zhang. Marlin: Two-phase bft with linearity. In *DSN*, pages 54–66. IEEE, 2022.

[37] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2013.

[38] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *PODC*, pages 347–356, 2019.