# Moonshot: Optimistic Proposal for Blockchain-Based State Machine Replication

v1.0 — Wednesday 12th July, 2023

Supra Research

***Abstract*—We introduce Moonshot, a new family of single-leader Byzantine Fault Tolerant (BFT) blockchain-based SMR protocols characterised by a new method of round pipelining facilitated by *optimistic proposal*. We formally describe Chained Moonshot, a variant of Moonshot that leverages the QC-chaining of Chained HotStuff while maintaining the best-case block finalisation latency of non-pipelined, vote-broadcast protocols like PBFT and Tendermint.**

**Chained Moonshot's combination of optimistic proposal and vote broadcasting ensure that validators are never idle by enabling proposal and voting to occur at network speed when the propagation times of Prepare and Proposal messages are equal. Despite Chained Moonshot's increased communication complexity in the Normal Path when compared to its recent predecessors, it demonstrably improves upon their block finalisation latency and block throughput. Our theoretical analysis reveals that Chained Moonshot has an expected 40% lower block finalisation latency and 50% decreased block period compared to Jolteon when the propagation times of Prepare and Proposal messages are equal. Our experimental results support this analysis, with Moonshot exhibiting an average of 41.1% lower block commit latency and 54.9% higher block throughput when compared to Jolteon in WANs of 10, 50, 100 and 200 nodes for varying payload sizes.**

***Index Terms*—blockchain consensus, state machine replication**

## I. Introduction

Public blockchain networks are revolutionising modern society by facilitating decentralised, immutable and verifiable data exchange for the first time in human history. These networks fundamentally provide decentralised computation and storage by marrying fault-tolerant distributed systems design with cryptography to enable higher transparency and accountability than traditional centralised computer networks.

At the heart of these networks are *consensus protocols* that enable *state machine replication* (SMR) [8]. A blockchain network is a form of distributed *state machine*, which is transitioned from one state to another by applying client-submitted instructions called *transactions*. SMR protocols ensure that every node in the network maintains a *consistent* state by facilitating their agreement upon the order in which these transactions should be executed. A *Byzantine Fault Tolerant* (BFT) [7] SMR protocol is one that tolerates a fixed number of faulty participants. These faulty processes, termed *Byzantine*, may crash or deviate arbitrarily from the protocol, but are assumed to be unable to break cryptographic primitives like signatures.

With these definitions in mind we distinguish between the term *blockchain network*, as the colloquial name for a network running any type of BFT SMR protocol, and *blockchain-based SMR protocol*, as the name for a particular category of SMR protocols. Blockchain-based SMR protocols differ from other types of SMR protocols in that they group transactions into *blocks*, with each new block committed by the network referencing the previously committed one as its *parent*, thus forming the blockchain.

We innovate *Moonshot*, a family of BFT blockchain-based SMR protocols characterised by *optimistic proposal*, a novel optimisation distinct from the round pipelining of Chained HotStuff [10] and its successors. This paper primarily focuses on *Chained Moonshot*, a variant of Moonshot that leverages both optimistic proposal and *QC chaining*, both of which techniques we go on to define properly in Section III.

### A. Paper Structure

We present Moonshot in several stages. Section II establishes the conceptual context for our new family of consensus protocols. Section III explains the fundamental insight behind Moonshot by informally comparing it with some of its predecessors, while Section IV describes Chained Moonshot in full along with the pseudocode. A more detailed discussion elaborating on some aspects of the design of Chained Moonshot and the properties of the protocol is presented in Section V. Section VI elaborates on how Chained Moonshot can be made more efficient by decreasing the size of Prepare and PrepareQC messages. Section VII provides formal proofs showing that Chained Moonshot achieves both the *Safety* and *Liveness* SMR properties. Section VIII presents an evaluation of our implementation against Jolteon. Finally, Section IX concludes the paper.

## II. Preliminaries

We now establish some definitions that will be used throughout the rest of the paper.

### A. Network Model

We consider a fully-connected network composed of a set $\mathcal{V} = \{v_1, ..., v_n\}$ of $n$ processes running a protocol $\mathcal{P}$ in the Authenticated Byzantine [4] setting. Accordingly, we assume the existence of an adversary that works to violate the guarantees of $\mathcal{P}$. We allow the adversary to corrupt up to $f$ processes when the network is initialised, which we thereafter refer to as being *Byzantine* and to the remainder as being *honest*. We assume that the adversary controls all communication channels, with the following caveats: We assume that the communication channels between the processes in $\mathcal{V}$ are perfect, meaning that messages sent by honest processes cannot be lost in transit and cannot be dropped by the adversary. We also assume that these channels are collectively partially synchronous.

We recall the definition of Partial Synchrony With GST established in [4], with some clarifications added in our own words to preserve the meaning of the definition per its original context:

**Definition 1.** *For every run $R$ of $\mathcal{P}$, there is a time $T$ such that $\Delta$ holds as an upper bound on the time taken for message delivery between each pair of honest processes $(v_i, v_j) \in \mathcal{V} \times \mathcal{V}$ in $[T, \infty)$. Such a time $T$ is called the Global Stabilisation Time (GST).*

We adopt a modified version of this definition to allow for alternating periods of asynchrony and synchrony during a given run $R$ of $\mathcal{P}$. Our updated definition aligns with the observation in [4] that $\Delta$ need not hold as the upper bound on message delivery forever after GST, but instead only until $GST + M$ (the original paper uses $L$, but we reserve this symbol for a future definition), where $M$ is the minimum duration of synchrony required for $\mathcal{P}$ to be guaranteed to make progress. We formalise this observation to allow us to give precise definitions of the properties that a blockchain-based SMR must exhibit, which we will come to shortly. Our modified version of partial synchrony follows.

We model each (possibly infinite) run of $\mathcal{P}$, denoted $R = ((A_0, S_0), (A_1, S_1), ...)$, as a sequence of pairs of finite time intervals with durations determined by the adversary. We allow the adversary to cause (any and) all communication channels to become asynchronous at their discretion during each $A_k$ ($k \geq 0$), allowing them to arbitrarily delay messages of their choice. We define the initial moment of each $S_k$ as the next Global Stabilisation Time, after which time the adversary must ensure that all protocol messages have an upper bound of $\Delta$ on their delivery latencies for the duration of the subsequent synchronous interval $S_k$. For the sake of simplicity, we assume that $\Delta$ is known to the designer of $\mathcal{P}$. We use $S_R$ to denote the set of all synchronous intervals in $R$.

We thus formally define Partial Synchrony With Repeated GST as follows:

**Definition 2.** *For every run $R = ((A_0, S_0), (A_1, S_1), ...)$ of $\mathcal{P}$, $\Delta$ holds as an upper bound on the time taken for message delivery between each pair of honest processes $(v_i, v_j) \in \mathcal{V} \times \mathcal{V}$ for the duration of each $S \in S_R$. We refer to the first moment of each $S \in S_R$ as a Global Stabilisation Time (GST).*

We know from [4] that no SMR protocol operating in a partially synchronous system of $n$ processes can tolerate $f > \lfloor \frac{n-1}{3} \rfloor$. For the ease of reading and without loss of generality, we fix $n = 3f + 1$ for the rest of the paper. Informally, we use the term *quorum* to refer to a subset of $\lfloor \frac{n+f}{2} \rfloor + 1$ (i.e. $2f + 1$ when $n = 3f + 1$) unique processes from $\mathcal{V}$, which is guaranteed to have an honest majority.

For the sake of this paper, we assume that $\mathcal{P}$ is a blockchain-based SMR protocol, which we now formally define.

### B. Blockchain-Based State Machine Replication

As mentioned in Section I, we informally define a blockchain-based SMR protocol as an SMR protocol in which client transactions are grouped into blocks that explicitly reference one another in order to form a blockchain. We assume that each block references at most one previously proposed block, and that these blocks are proposed in sequential *rounds* by an elected *leader* process. We assume that every process $v \in \mathcal{V}$ that participates in such a protocol maintains a local copy, denoted $\mathbf{B_v}$, of the *canonical blockchain*. We use $\mathbf{B_{v_i}} \preceq \mathbf{B_{v_j}}$ to denote that the canonical blockchain of $v_i$ is a prefix of that of $v_j$.

With these definitions in mind, we now formally define the properties of a blockchain-based SMR protocol $\mathcal{P}$ operating in the partially synchronous setting.

Let $\mathcal{R}_{\mathcal{P}}$ denote the set of all possible runs of $\mathcal{P}$. Additionally, let $C_R(M) = \{S \mid S \in S_R \text{ and } |S| \geq M\}$, where $|S|$ denotes the duration of the given synchronous interval $S$ of the run $R$ and $M$, as before, is the minimum duration of synchrony required for $\mathcal{P}$ to be guaranteed to make progress. We observe that since $\mathcal{P}$ is a blockchain-based SMR protocol, $\mathcal{P}$ *makes progress* when each honest $v \in \mathcal{V}$ adds at least one new block proposed by an honest leader to its local blockchain $\mathbf{B_v}$.

**Definition 3.** *In a partially synchronous network of $n$ processes with an $f$-limited adversary, $\mathcal{P}$ satisfies the following properties:*

**Liveness.** *For every run $R \in \mathcal{R}_{\mathcal{P}}$, for each synchronous interval $S \in C_R(M)$, each honest process $v \in \mathcal{V}$ appends at least $\lfloor \frac{|S|}{M} \rfloor$ new blocks proposed by honest leaders to its local blockchain $\mathbf{B_v}$ during $S$.*

**Safety.** *For every run $R \in \mathcal{R}_{\mathcal{P}}$, for each pair of honest processes $(v_i, v_j) \in \mathcal{V} \times \mathcal{V}$, at each moment during $R$ either $\mathbf{B_{v_i}} \preceq \mathbf{B_{v_j}}$ or $\mathbf{B_{v_j}} \preceq \mathbf{B_{v_i}}$.*

## III. Insight

We now discuss the core insight behind Moonshot, but before we do so we first establish a model for comparing Moonshot to its predecessors.

### A. Method of Analysis

We analyse the theoretical performance of Moonshot and compare it to its predecessors in terms of its block period and block commit latency. We define block period in terms of the network delay between consecutive block proposals and commit latency as the delay between the proposal of a block and its commit by the $2f + 1$th process. We use $\delta$ to represent the average message transmission latency after GST. We measure $\delta$ over the duration of any given run of the related protocol as the average time between the dispatch of any message on a point-to-point link after GST until its receipt. Accordingly, $\delta \leq \Delta$. We observe that $\delta$ is imprecise since it does not factor in either the relative sizes of the different types of messages or the size of the network, but consider it to be a suitable approximation for now. We will go on to define a more precise analytical model in a later version of this paper, which will introduce additional variants of Moonshot.

### B. Contribution

The Practical Byzantine Fault Tolerance (PBFT) protocol [3] was the first workable solution for BFT SMR in the partially synchronous setting. Later, Tendermint [2] adapted PBFT for the blockchain setting. Figure 1 shows the normal case operation of the Tendermint protocol, where the Prevote and Precommit phases have been renamed to Prepare and Commit, respectively, for the convenience of relating to the terminology used in this paper. Each Tendermint instance proceeds in three phases: *Propose*, *Prepare* and *Commit*.

Communication in Tendermint is achieved via a gossip protocol in the original construction of the protocol, but for the sake of a fair comparison to our protocol we assume a modified version of Tendermint that operates in the same setting as described in Section II. Accordingly, we consider a variant of Tendermint that sends all messages over point-to-point links in a fully-connected network. We observe that since Tendermint requires that "if a correct process p receives some message m at time t, all correct processes will receive m before $max\{t, GST\} + \Delta$" [2] for the sake of liveness, honest processes must re-broadcast all messages that they receive.

In the first phase of Tendermint, the leader of the current round broadcasts a block in a signed Proposal message. A validator then enters the Prepare phase after receiving a valid Proposal from the leader, and re-broadcasts it along with a Prepare message to indicate its endorsement of the Proposal. The validator then waits to receive a quorum of valid Prepare messages (each of which it must re-broadcast) and then constructs a *Prepare Quorum Certificate* (Prepare QC) as verifiable proof that a quorum of processes have accepted the leader's proposal. After forming this QC, the process enters the Commit phase and broadcasts a Commit message as a second endorsement of the Proposal. As before, the process then waits to receive a quorum of valid Commit messages before forming *Commit Quorum Certificate* and committing the block. Accordingly, this variant of Tendermint has a best-case proposal-to-commit latency, a best-case block period of $3\delta$ and a communication complexity of $O(n^3)$.

Chained HotStuff [10] introduced the notion of *round pipelining*. For the rest of the paper we distinguish between *pipelining*, as a methodology of concurrently transmitting messages, and *chaining*, as a mechanism enabling a message to serve multiple purposes. Chained HotStuff enables multiple consensus rounds to proceed concurrently by allowing leaders to create new Proposals

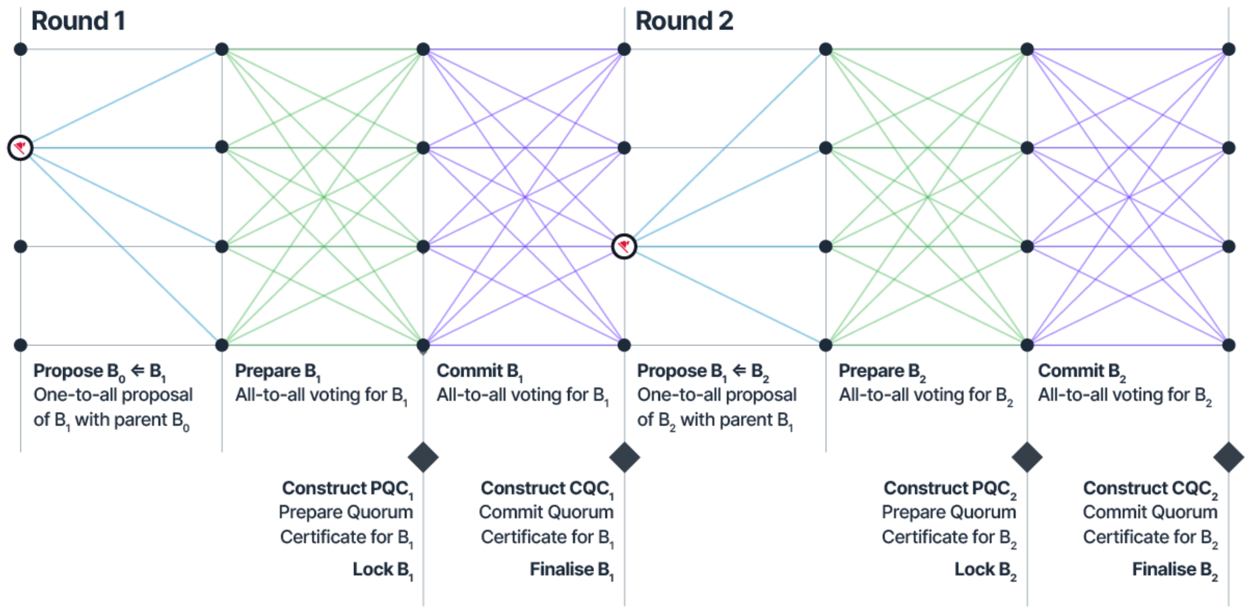



Fig. 1. Tendermint Normal Path

justified by the Prepare QC for the proposal of their predecessor. The original protocol assumes that leaders serve as aggregators for the Prepare votes for their own Proposals, resulting, like Tendermint, in a best-case block period of $3\delta$. DiemBFT [9] and Jolteon [6], both of which are variants of Chained HotStuff, instead have validators relay their Prepare votes directly to the next leader and thus exhibit a best-case block period of of $2\delta$, a noteworthy improvement over Tendermint.

We take Chained HotStuff's round pipelining one step further to innovate *Moonshot*, a new class of blockchain-based SMR protocols with a best-case block period of $\delta$. We start by observing that the liveness of blockchain-based SMR protocols depends on a leader being able to refer to the block proposed for the previous height when making its own Proposal, meaning that the Propose phases of successive heights must proceed sequentially. However, the safety of such protocols is dictated only by their rules for voting and committing. Therefore, the Propose phase for a given height, despite being dependent on the Propose phases of the previous heights for liveness, is actually independent of their corresponding Prepare phases with respect to safety. This observation reveals a new avenue for pipelining wherein the Prepare phase of an earlier round can be safely overlapped with the Propose phases of later ones. Allowing leaders to extend blocks not yet accepted by a quorum improves bandwidth utilisation under normal-case operation by enabling Prepare votes and new Proposals to be transmitted concurrently. Hereafter, we refer to this type of pipelining as *optimistic proposal*, and refer to traditional pipelining as *round pipelining*.

Chained HotStuff and its derivatives suffer an increased commit latency compared to Tendermint as a result of their pursuit of linear normal-case communication complexity. They achieve this by using a single aggregator to collect votes, which necessarily increases the minimum duration of each voting phase from $\delta$ to at least $2\delta$. Consequently, Jolteon, a two-chain variant of Chained HotStuff (i.e. a variant with two voting phases per block) and the most efficient of the aforementioned derivatives of Chained HotStuff, exhibits a best-case commit latency of $5\delta$.

For the purposes of this paper, we assume that trading increased communication complexity for decreased best-case theoretical block period and commit latency will produce a practically more efficient protocol. Accordingly, we present *Chained Moonshot*, a member of the Moonshot family of protocols that utilises QC chaining and vote-broadcasting to obtain a best-case block period of $\delta$, commit latency of $3\delta$ and communication complexity of $O(n^2)$, as shown in Figure 2.

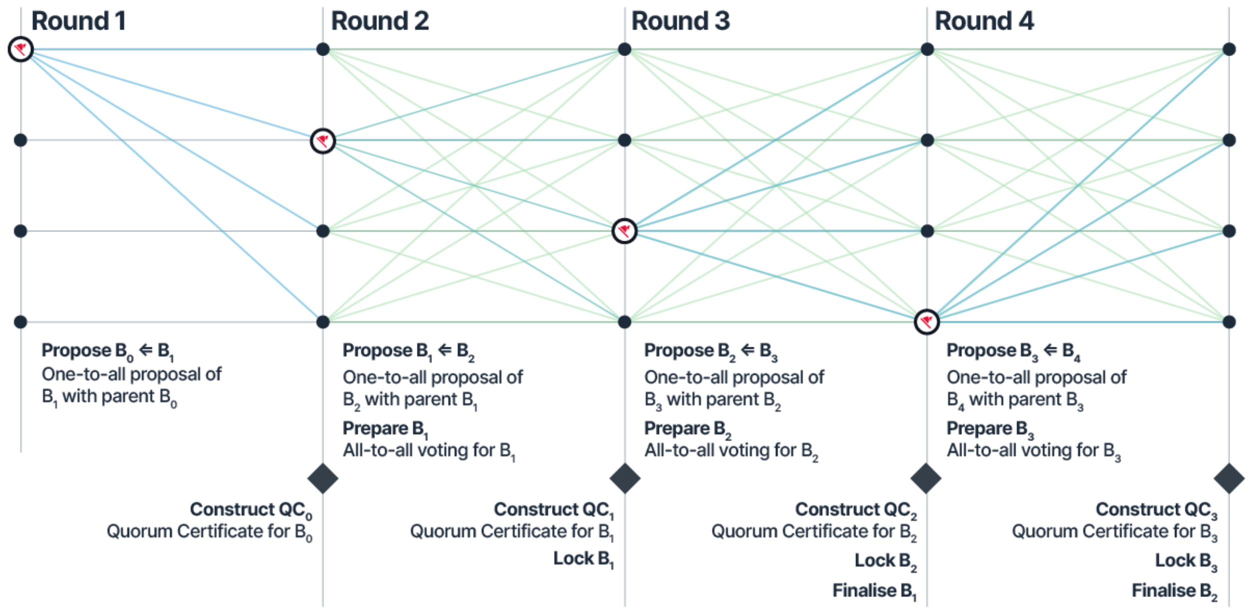



Fig. 2. Chained Moonshot Normal Path

## IV. Chained Moonshot

We now present the Chained Moonshot SMR protocol.

### A. Blockchain Model

A well-formed Chained Moonshot block $B_r$ contains the round identifier $r$ for which the block was proposed, the digest of its parent block $B'$, and a possibly-empty payload. We model this payload as an abstract representation of a totally-ordered set of transactions from the clients of the system. We consider the problems of transaction delivery and execution to be orthogonal to the problem of transaction ordering and so do not require payloads to include the transactions themselves. We assume that the method of transaction delivery utilised by any implementation of Moonshot guarantees that every transaction that is ordered by Moonshot is eventually delivered to all honest processes. We likewise assume that the method of execution respects the total ordering imposed upon the transactions by the blockchain, and that the transactions themselves contain only deterministic operations.

We model the local blockchain of each process $v \in \mathcal{V}$, $\mathbf{B}_v$, as a totally-ordered sequence of blocks $\mathbf{B}_v = (B_0, B_r, \ldots, B_h)$, indexed by the rounds for which they were proposed. We assume that every $\mathbf{B}_v$ is initialised with $B_0$, a common Genesis Block that contains all configuration information required by the system at start-up, about which we remain agnostic. We assume that every $v \in \mathcal{V}$ also possesses $QC_0$, a Quorum Certificate justifying $B_0$, upon initialisation.

### B. Leader Election

Being a blockchain-based SMR protocol, Chained Moonshot proceeds in rounds. Each $v \in \mathcal{V}$ is assigned the role of either *leader* or *validator* upon entering a new round. The leader of each round, $L_r$, is elected via a function $L$ that we assume is *fair*, giving every process in the system equal opportunity to become leader. We distinguish between *deterministic fairness* and *probabilistic fairness* in Definition 4.

**Definition 4.** *A leader election function $L$ that samples from $\mathcal{V}$ can be said to be fair if it satisfies either of the following definitions:*

**Deterministic Fairness.** *A deterministically fair $L$ guarantees that there exists some $k$ such that each $v \in \mathcal{V}$ leads exactly $k$ rounds every $kn$ rounds.*

**Probabilistic Fairness.** *A probabilistically fair $L$ guarantees that there exists some $k$ such that the expected number of rounds led by each $v \in \mathcal{V}$ every $kn$ rounds is $k$.*

We observe that since the definition of Liveness given in Definition 3 requires at least one honest block proposal to be committed during each synchronous interval lasting at least $M$, the leader election function of $\mathcal{P}$ must deterministically elect at least one honest leader during this time (notice that this is a necessary condition for Liveness, but is not sufficient). Consequently, any $\mathcal{P}$ that uses a probabilistic $L$ can only be said to obtain *probabilistic liveness* under this definition.

We make no assumptions about whether $L$ can be used to predict leaders in advance and observe that there are known strategies for preventing this (e.g. [5]) when it is considered undesirable. We also assume that $L_r$ has access to a pool of unique, uncommitted transactions or abstractions thereof that it samples from when creating a new block, but leave the related selection function abstract.

### C. Specification

We simplify the following specification by making a key assumption to eliminate complexity from the protocol that is orthogonal to our contribution. Namely, we assume that both Prepare messages and QCs contain the related block. We observe that this results in a protocol with significantly higher communication costs than the standard approach for such messages, which is to instead include a unique identifier called the *digest* of the block. However, the methods for implementing this optimisation are well-known, so we predicate our subsequent analyses and proofs on the variant of Chained Moonshot that implements this optimisation. We discuss this variant in greater detail in Section VI.

Table I shows the variables that each process $v$ is required to maintain according to the pseudocode presented in Algorithms 1 and 2. We also assume the availability of the functions defined in Table II, for which we provide only abstract definitions.

We present our pseudocode for Chained Moonshot as a series of event handlers of the form $upon\ \langle event\rangle\ do\ \langle action\rangle$. We use the following qualifiers to differentiate between the different types of events that trigger the processing of protocol messages:

- We use the term $observing$ to indicate that the event's validity condition is independent of $v$'s current round, implying that it need not persist the corresponding message unless the subsequent action causes it to do so.
- We use $posessing$ to indicate that $v$ will need to enter a particular round in order to satisfy the event's validity condition, which will require $v$ to persist the corresponding message if it receives it before entering that round.

TABLE I
LOCAL VARIABLES FOR $v \in \mathcal{V}$

| | |
|---|---|
| $a_f$ | The highest round for which this process has accepted (and thus broadcasted a Prepare message for) a Fallback Recovery Proposal. Initially 0. |
| $a_n$ | The highest round for which this process has accepted (and thus broadcasted a Prepare message for) a Normal Proposal. Initially 0. |
| $B_n$ | The last block that this process proposed as a Normal Proposal. Initially $B_0$. |
| $B_h$ | The block most recently appended to $\mathbf{B_v}$. Initially $B_0$. |
| $\mathbf{B_v}$ | A representation of all blocks committed by $v$. Initially covers only $B_0$. |
| $E$ | A set containing the identifiers of all rounds that $v$ considers to have expired due to having sent the corresponding Timeout message. Initially $\emptyset$. |
| $id$ | The public identifier of $v$. |
| $p_f$ | The highest round for which this process has broadcasted a Fallback Recovery Proposal. Initially 0. |
| $qc_l$ | The currently locked QC. Initially $QC_0$. |
| $r_c$ | The identifier of the current round. Initially 1. |
| $t_r$ | The timer used to trigger Timeout events. Initially 0. |
| $U$ | A set containing all uncommitted QCs observed by $v$. Initially $\emptyset$. |

We use no qualifier for the handler for the expiry of the round timer $t_r$ because it does not process any protocol message.

These event handlers in turn make use of *procedures*, by which we abstract certain functionality for ease of reading. Likewise, we use the following symbols to that same end: We use $|$ in place of the term "such that", and $\wedge$, $\neg$, $\leftarrow$ and $=$ as the Logical And, Logical Negation, Assignment and Equality operators, respectively. We also use $\forall$ to denote the universal quantifier and replace object fields with _ when they are not used by the handler in question. We use $B \Longleftarrow B'$ (pronounced "$B'$ extends $B$") to denote that $B$ is the parent of $B'$, and $B \Longleftarrow^* B''$ to indicate that $B$ is an ancestor of $B''$. More formally, we use $\Longleftarrow^*$ to denote the reflexive and transitive closure of $\Longleftarrow$.

#### Message Definitions

Our protocol proceeds via the messages *Normal Proposal*, *Fallback Recovery Proposal*, *Prepare*, *Timeout*, *QC* and *TC*. To keep the pseudocode of Algorithms 1 and 2 concise, we use the abbreviations *N*, *F*, *P* and *T* for the first four messages, respectively. The contents of these messages and their respective validity conditions are described below. As mentioned in Section II, Chained Moonshot operates in the Authenticated Byzantine setting, so we assume that all messages are signed by the

TABLE II
ABSTRACT FUNCTIONS

| | |
|---|---|
| $broadcast(m)$ | Sends the message $m$ to all $v \in \mathcal{V}$. |
| $cleanup(r)$ | Purges all Prepare and Proposal messages for $r' \leq r$ from memory and disk. Also sets a timer for $2\Delta$ and purges all Timeout messages for $r' \leq r$ including those in $E$ upon its expiry. |
| $commit(qc)$ | Schedules $qc.B$ and all of its uncommitted ancestors to be appended to $v$'s local blockchain in increasing order of their height as they become available. That is, if $v$ has yet to receive some ancestor $B_a$ of $qc.B$, then it waits until $B_a$ it has received $B_a$ before appending any of its descendents. Upon appending $B_r$, $v$ removes all QCs and blocks for $r' \leq r$ from $U$ and any other in-memory or on-disk storage, except for from $B_h$, $\mathbf{B_v}$, $B_n$ and $qc_l$. We leave it up to the implementer as to whether this procedure also executes the transactions included in $B$, which we observe may be done later if the system so requires. |
| $digest(m)$ | Returns a fixed, concise, collision-resistant representation of $m$. |
| $hasQuorum(m)$ | Returns $true$ only if $m$ is a QC or a TC, $m.c$ was constructed from a quorum of valid component messages for $m$ and all of these messages were sent by different members of $\mathcal{V}$. |
| $isMaxQC(qc, c)$ | Returns $true$ only if $c$ proves that $qc.r$ is the greatest round number of any QC used to construct $c$. |
| $max(v_1, v_2)$ | Returns $v_1$ if $v_1 > v_2$, otherwise returns $v_2$. |
| $maxQC(s)$ | Returns the QC with the highest round included in the set $s$. |
| $resetRoundTimer()$ | Starts the round timer $t_r$ or resets it if it is already running. |
| $txs()$ | Returns a set of transactions that have not been included in a committed block, or $\emptyset$ if none are currently available. |
| $sendersAreUnique(s)$ | Returns true if all messages contained in the set $s$ were sent by different members of $\mathcal{V}$. |

sender and come with all of the information required to verify this signature. We further assume that these signatures cover a domain-unique identifier and the round number in addition to the value being signed, to prevent replay attacks.

A well-formed Normal Proposal contains the proposed block $B_r$, which, as previously mentioned, in turn contains at least the round identifier $r$, the digest of the parent block $B'$, and a possibly-empty set of transactions. Including the digest of $B'$ in $B_r$ in this manner allows us to implement the parent relation $B' \Longleftarrow B_r$. Comparatively, a well-formed Fallback Recovery Proposal contains the proposed block $B_r$ and a well-formed Timeout Certificate for $r-2$, denoted $TC_{r-2}$. We elaborate on the validity conditions for each proposal type in the following subsections.

A well-formed Prepare vote contains the related block $B_r$. Since honest processes are allowed to vote up to twice in a given round $r$, a validator $v$ with $r_c \leq r+1$ accepts at most two Prepare messages for round $r$ from each of its peers to prevent the Byzantine processes from consuming its memory by spamming votes for the same round. We omit this logic from the pseudocode presented in Algorithm 1 for the sake of brevity.

A well-formed QC for a block $B_r$, denoted $QC_r$, contains $B_r$ and a representation $c$, chosen by the implementer, of a quorum of valid Prepare messages for $B_r$. A validator $v$ considers $QC_r$ valid only if $c$ proves that $QC_r$ was constructed from a quorum of valid Prepare messages for $B_r$ from unique processes.

A well-formed Timeout message for a round $r$, denoted $T_r$, contains the related round identifier $r$ and a QC. This QC should be the highest QC observed by the sender at the time that it created $T_r$. A validator $v$ considers $T_r$ valid if it is the first such message from the given sender and $v$ has not yet garbage-collected its Timeout messages for $r$ per the semantics of the `cleanup` function defined in Table II. As with the accept logic for Prepare messages, we omit this logic for Timeout messages from the pseudocode presented in Algorithm 2 for the sake of brevity.

A well-formed $TC_r$ contains the round identifier $r$ and a certificate $c$ constructed from a quorum of valid Timeout messages for $r$ from unique processes. It also contains $qc'$, the QC with the maximum round number contained in the set of Timeout messages used to construct $c$. The certificate $c$ must prove both that a quorum of processes have sent $T_r$ messages and that $qc'$ is the maximum QC submitted by the quorum. We observe that one valid construction for $c$ is the set containing the pairs $(\sigma_v(r, qc.r), qc.r)$ derived from the original quorum of $T_r$ messages, where $\sigma_v(r, qc.r)$ is the signature of the sender $v$ on both $r$ and the round number of the QC that it included in its $T_r$. A validator $v$ considers $TC_r$ valid only if $c$ proves the aforementioned properties and $qc'$ is for a round less than $r$.

We observe that the above definitions admit some abuse from the adversary, especially if it is able to predict the rounds in which it will have the right to propose in advance. Specifically, without additional validity conditions for Normal Proposals, Prepares and Timeouts, the adversary can spam honest processes with valid messages for higher rounds until their storage is consumed. We consider this problem orthogonal to our contribution, but observe that one way that this could be prevented without increasing the asymptotic communication complexity of the protocol is by requiring these messages for a given round $r$ to include a quorum threshold signature on $r-2$.

### D. Normal Path for $v \in \mathcal{V}$

All processes start in round 1 in the state described in Table I upon the initialisation of Chained Moonshot. If $L_1$ is honest then it attempts to create and broadcast $B_1$, a child of $B_0$, as a Normal Proposal. Subsequently, all honest processes immediately advance to round 2 via $QC_0$ and reset their round timers. We omit this sequence from the pseudocode given in Algorithm 1 for the sake of brevity.

More generally, an honest process $v$ enters round $r+2$ via the *Normal Round Transition Rule* and resets its round timer after observing a QC for $r$. After entering round $r$, $L_r$ becomes eligible to propose a Normal Proposal and $v$ becomes eligible to vote for proposals from $L_{r-1}$. Specifically, the first time that $L_r$ possesses a Normal Proposal containing the block $B_{r-1}$ with parent $B'$ from $L_{r-1}$ whilst in $r$, it tries to create a Normal Proposal of its own. If $L_r$ has not already created a proposal for $r$, then it will succeed and will create and broadcast a Normal Proposal containing a block $B_r$ that extends $B_{r-1}$.

Notice that $L_r$ does not verify the certification of either $B_{r-1}$ or its parent before making its own proposal. Instead, it proposes optimistically, assuming that both blocks will become valid if they are not already, and that their certificates will be locked by a quorum of its peers. If $L_{r-1}$ is Byzantine it can therefore take advantage of this fact to cause $L_r$ to create an invalid Normal Proposal by sending it an arbitrary block for $r-1$. However, $L_r$ will usually eventually be able recognise that it has been lied to and will update its proposal accordingly, as we will see.

Comparatively, $v$ is not permitted to vote for $B_{r-1}$ unless it is locked on a QC for $B'$. This helps the protocol to obtain the Safety SMR property, as elucidated in Section VII. In addition to being locked on the QC for $B'$, $v$ must not yet have voted for a Normal Proposal from $L_{r-1}$, must not have sent $T_r$ and $B'$ must have been proposed for $r-2$. If all of these conditions are satisfied then $v$ considers $B_{r-1}$ to have satisfied the *Normal Vote Rule* and broadcasts a Prepare vote for $B_{r-1}$.

After observing a quorum of Prepare votes for $B_{r-1}$ from unique processes including itself, $v$ constructs a QC for $B_{r-1}$ by aggregating the Prepare votes into a verifiable proof that a quorum of processes voted for $B_{r-1}$.

**Algorithm 1:** Normal Path for $v \in \mathcal{V}$

**procedure** `proposeNormal(`$B$`)` **do**
  **if** $L_{r_c} = id \wedge r_c > p_f \wedge \neg(B \Longleftarrow B_n)$ **then**
    $B_n \leftarrow$ `Block(`$r_c$`, digest(`$B$`), txs())`
    `broadcast(N(`$B_n$`))`
**procedure** `advanceToRound(`$r$`)` **do**
  **if** $r > r_c$ **then**
    $r_c \leftarrow r$
    `resetRoundTimer()`
    `cleanup(`$r-2$`)`
**procedure** `tryLock(`$qc$`)` **do**
  $r \leftarrow qc.B.r$
  **if** $r > qc_l.B.r \wedge r+1 \geq r_c \wedge r \notin E$ **then**
    $qc_l \leftarrow qc$
**procedure** `tryCommit(`$qc$`)` **do**
  **forall** $qc' \in U$ **do**
    **if** $qc'.B.r + 1 = qc.B.r$
     $\wedge\ qc.B \Longleftarrow qc'.B$ **then**
      `commit(`$qc'$`)`
    **if** $qc.B.r + 1 = qc'.B.r$
     $\wedge\ qc'.B \Longleftarrow qc.B$ **then**
      `commit(`$qc$`)`
**upon** *first observing* $qc \leftarrow$ `QC(`$B$`, _)` *either received in a valid protocol message or built from a quorum of* `P(`$B$`)`
  | `hasQuorum(`$qc$`)`
**do**
  **if** $B.r + 2 > r_c$ **then**
    `proposeNormal(`$B$`)`
  $U \leftarrow U \cup \{qc\}$
  `tryLock(`$qc$`)`
  `tryCommit(`$qc$`)`
  `broadcast(`$qc$`)`
  `advanceToRound(`$B.r+2$`)`
**upon** *first possessing a* `N(`$B$`)` *from* $L_{r_c - 1}$
  | $B.r + 1 = r_c$
  $\wedge\ r_c > B_n.r$
**do**
  `proposeNormal(`$B$`)`
**upon** *first possessing a* `N(`$B$`)` *from* $L_{r_c - 1}$
  | $B.r + 1 = r_c \wedge B.r > a_n \wedge B.r \notin E$
  $\wedge\ qc_l.B \Longleftarrow B$
  $\wedge\ qc_l.B.r + 1 = B.r$
**do**
  `broadcast(P(`$B$`))`
  $a_n \leftarrow B.r$

We remain agnostic as to what this proof consists of for the sake of the generality this specification, but discuss some options for its construction in the Section V.

The first time that $v$ observes a valid $QC_r$ it executes the *QC Processing Rule*. This happens when $v$ either constructs $QC_r$ itself or receives it from one of its peers in a protocol message. Importantly, if $v$ receives $QC_r$ in another protocol message then it processes $QC_r$ before the message that contains it. Firstly, $v$ checks the round of the QC. If $v$ is currently in round $r+1$ then it attempts to propose via the *QC Extension Rule*. This attempt only succeeds if $v$ is $L_{r+1}$, has not yet created a Fallback Recovery Proposal for $r+1$ and has not yet created a Normal Proposal extending the certified block. Importantly, this rule allows $L_{r+1}$ to re-propose if it originally created an invalid Normal Proposal due to receiving an equivocal or invalid Normal Proposal from $L_r$.

Subsequently, $v$ adds $QC_r$ to its set of uncommitted QCs before attempting to lock it. The *Lock Rule* only allows $v$ to lock $QC_r$ if $r$ is greater than the round of its currently locked QC, it is in $r+1$ or lower and has yet to send $T_r$. Intuitively, since $v$ is only allowed to vote for a Normal Proposal if it is locked on its parent, the final requirement ensures that if $f+1$ honest processes send Timeout messages for $r$ then it will be impossible for any honest process to observe a QC for a Normal Proposal for $r+1$. This helps to ensure the Safety of the protocol, the complete proof for which is given in Section VII.

After trying to lock $QC_r$ $v$ checks whether it authorises the commit of any new blocks. It does this by checking whether its set of uncommitted QCs contains a QC for a block $B'$ such that either $B' \Longleftarrow B_r$ and $B'.r = r-1$ or $B_r \Longleftarrow B'$ and $B'.r = r+1$. In the former case, $QC_r$ triggers the *Commit Rule* for $B'$ and in the latter the QC for $B'$ does so for $B_r$. This check prevents the adversary from delaying commits by delivering QCs out of order. In both cases, $v$ schedules the committed block to be appended to $\mathbf{B_v}$ once it has received all of its ancestors.

Next, $v$ broadcasts $QC_r$ to ensure that all of its peers will observe it in a timely manner. This is necessary to ensure the Liveness of the protocol, which the adversary can otherwise inhibit, as discussed in Section V. Finally, $v$ attempts to enter $r+2$ as previously described.

In addition to the aforementioned actions that $v$ takes upon entering round $r$ it also becomes free to purge all Prepare and Proposal messages for $r' \leq r-2$, but must continue to accept previously unseen QCs for lower rounds. We include this action as the default behaviour in

**Algorithm 2:** Fallback Path for $v \in \mathcal{V}$

```
procedure proposeFallback(tc) do
  r := tc.r + 1
  if L_r = id then
    d ← digest(tc.qc'.B)
    B ← Block(r, d, txs())
    broadcast(F(B, tc))
    p_f ← r
procedure timeout(r) do
  if r ∉ E then
    broadcast(T(r, maxQC(U)))
    E ← E ∪ {r}
upon t_r = τ
do
  timeout(r_c − 1)
upon first observing a set S of f + 1 T(r, _)
  | sendersAreUnique(S)
  ∧ r > B_h.r
  ∧ ∀qc ∈ U, qc.r ≠ r
do
  timeout(r)
upon first observing tc ← TC(r, qc', c) either
    received in a valid F or built from a quorum of
    T(r, _)
  | hasQuorum(tc)
  ∧ isMaxQC(qc', c)
do
  if r + 2 > r_c then
    proposeFallback(tc)
  timeout(r)
  advanceToRound(r + 2)
upon first possessing a F(B, tc) from L_{r_c−1}
  with qc' = tc.qc'
  | B.r + 1 = r_c ∧ B.r = tc.r + 1
  ∧ qc'.B ⟸ B
  ∧ hasQuorum(tc)
  ∧ isMaxQC(tc.qc', tc.c)
do
  proposeNormal(B)
  if B.r > a_f ∧ B.r ∉ E then
    broadcast(P(B))
    a_f ← B.r
```

our pseudocode, abstracted by the `cleanup` function, but recognise that some implementations may wish to preserve this information for auditing or other purposes.

### E. Fallback Path for $v \in \mathcal{V}$

Suppose that an honest process $v$ enters a new round $r$ at time $t$. It subsequently enters the fallback path and broadcasts $T_{r-1}$ if it fails to either observe a QC for any block proposed in $r-1$ or enter a higher round, before $t+\tau$, where $\tau > 4\Delta$. When it broadcasts $T_{r-1}$ as a result of this condition or as a result of any of the following rules, then it adds $r-1$ to its set of expired rounds, ignores any subsequently received proposals from $L_{r-1}$ and accepts but does not lock any related QC.

As mentioned in the message definitions, an honest process that has yet to garbage collect the Timeout messages for $r$ per the `cleanup` function accepts the first Timeout message for $r$ that it receives from a given sender. If $v$ observes $f+1$ valid $T_r$ messages from unique senders including itself while having yet to append a block for $r$ or higher to $\mathbf{B_v}$ and not having $QC_r$ in $U$, then it triggers the *Timeout Sync Rule* and broadcasts its own $T_r$ if it has not already done so. If $v$ waits at least $2\Delta$ after entering $r+2$ before garbage collecting Timeout messages for $r$ or lower, which we assume is facilitated by the aforementioned `cleanup` function, then this rule provides an important guarantee. Namely, that after GST all honest process will enter to $r+2$ or higher within $2\Delta$ of the first honest process entering $r+2$, regardless of the behaviour of the adversary, as shown in Lemma 7.

After observing a quorum of valid $T_r$ messages, $v$ constructs $TC_r$ as previously described. The first time $v$ observes such a TC, be it due to constructing the TC itself or due to receiving it in a Fallback Recovery Proposal, it executes the *TC Processing Rule*. As with QCs, $v$ executes this rule for any TCs that it receives in Fallback Recovery Proposals before processing the related proposal.

Initially, if $v$ is $L_{r+1}$, is in a round below $r+2$ and has yet to create a Fallback Recovery Proposal, then it creates a new Fallback Recovery Proposal containing a new block $B' \Longleftarrow B_{r+1}$, where $B'$ is the block certified by $TC_r.qc'$. Like the QC Extension Rule, this rule allows $L_{r+1}$ to correct any Normal Proposal that it may have made as a result of previously observing a Normal Proposal from $L_r$, which it can infer is now guaranteed to fail. As previously observed, $v$'s observation of $TC_r$ makes this inference possible because of the Lock and Normal Vote Rules (see Lemma 2). Subsequently, $v$ broadcasts $T_r$ if it has yet to do so and enters $r+2$ if it has yet to enter $r+2$ or higher.

Any validator in round $r$ that possesses a Fallback Recovery Proposal $F(B_{r-1}, TC_{r-2})$ from $L_{r-1}$ ensures that the parent of $B_{r-1}$ is the block certified by $TC_{r-2}.qc'$. The first time it possess such a proposal it attempts to create a new Normal Proposal extending $B_{r-1}$, succeeding only if it is $L_r$. It then attempts to send a Prepare vote for $B_{r-1}$, succeeding only if it has not yet voted for a Fallback Recovery Proposal for $r-1$ or sent $T_{r-1}$.

## V. Discussion

We now elaborate on Chained Moonshot's design.

### A. Asynchronous Agreement

We observe that Chained Moonshot's rules for voting, locking and committing enable processes to participate in consensus without possessing the full blockchain. This makes it particularly useful for systems that dynamically change the membership of $\mathcal{V}$. It likewise makes Chained Moonshot well-suited for application in systems that decouple transaction ordering, delivery and execution. For instance, this property improves the performance of the optimised protocol discussed in Section VI as it allows the block synchronisation subprotocol and consensus to run in parallel when ordering is decoupled from execution.

### B. Externally Verifiable Blockchain

An *externally verifiable* blockchain is one that enables processes outside of the validator set that know the membership of this set and the public keys of its constituents to verify that a given block is a part of the canonical blockchain by way of a *Commit Certificate*. We propose two different constructions for Chained Moonshot Commit Certificates.

If the implementation of Chained Moonshot persists all finalised blocks and their QCs then a validator can construct a Commit Certificate for $B_r$ without any modifications to the protocol by aggregating:

1) The blocks forming the subsequence of the blockchain from $B_r$ to $B_s, B_{s+1}$ such that:
   - $B_s$ and $B_{s+1}$ are proposed in rounds $s$ and $s+1$ respectively,
   - $s \geq r$, and;
   - $B_s$ is the parent of $B_{s+1}$.
2) The QC for $B_{s+1}$.

This construction proves that $B_r$ is an ancestor of a block that satisfied the Chained Moonshot Commit Rule, namely $B_s$.

Alternatively, the system could generate Commit Certificates by performing an additional round of $f+1$-threshold agreement for each finalised block. For example, a validator could broadcast its signature on each block that it commits (along with any other data that the implementer wishes to use to distinguish this message from a Prepare message) and aggregate $f+1$ such signatures to form a Commit Certificate. Under this construction, the Commit Certificate guarantees that at least one honest process has committed the related block, so the Safety and Liveness properties of the system ensure that every other honest process will eventually do the same.

### C. QC Broadcasting

As mentioned in Section IV, Chained Moonshot requires processes to broadcast all QCs that they observe in order to preserve Liveness. Without this rule, the current protocol would otherwise be vulnerable to the following attack from the adversary:

Suppose that all honest processes are in round $r$, having entered $r$ via $QC_{r-2}$. Suppose also that $L_{r-1}$ was Byzantine and multicasted a valid Normal Proposal to only $f+1$ honest processes. These $f+1$ honest processes therefore will vote for the related $B_{r-1}$, giving the adversary control over $QC_{r-1}$. The adversary can then selectively deliver this QC to a set $A$ of up to $f$ honest processes, causing them to enter $r+1$ while the remainder, say $B$, remain behind in $r$. Subsequently, if the adversary continues to withhold $QC_{r-1}$ from them then $B$ will eventually send $T_{r-1}$. However, if this occurs more than $2\Delta$ before GST then the processes in $A$ will garbage collect their Timeout messages for $r-1$ and will no longer send $T_{r-1}$ upon observing the $f+1$ or more $T_{r-1}$ messages from their peers in $B$. Consequently, if the Byzantine processes remain silent, then $B$ will remain permanently stuck in $r$.

This attack can be mitigated by removing the garbage collection logic for Timeout messages from the protocol. However, removing this logic would require a different proof for Lemma 7, which would in turn affect later lemmas also. Without both garbage collection and QC broadcasting, the lower bound on $\tau$ derived in Section VII would likely increase and it is not immediately clear that Liveness would remain intact. Moreover, we initially added these rules in the hope of arriving at a protocol with bounded memory requirements. We are still exploring this possibility so do not consider removing them to be a useful solution at this time. We will update this paper with our findings when we have completed the relevant proofs.

### D. Fallback Recovery

As mentioned in Section III, Moonshot's key innovation is optimistic proposal, which allows the Prepare and Propose phases of successive rounds to proceed in parallel under normal conditions. This necessarily requires leaders to propose optimistically and assume that the proposal of their predecessor will become certified. However, this is not guaranteed. The adversary can leverage its control of the network and the Byzantine processes to cause a leader to propose optimistically whilst preventing the certification of the parent of its proposal and by extension, of the proposal itself.

Chained Moonshot neutralises this attack vector by allowing a leader to replace its optimistic Normal Proposal with a fully-justified Fallback Recovery Proposal. This is facilitated by the Chained Moonshot Lock and Normal Vote rules. The former ensures that if an honest process sends $T_r$ then it does not lock $B_r$ and the latter that if an honest process is not locked on $B_r$ then it will not vote for $B_{r+1}$ if $B_r \Longleftarrow B_{r+1}$. Consequently, if $L_{r+1}$ observes $TC_r$ after optimistically such a $B_{r+1}$, then it can infer that at least $f+1$ honest processes must not have observed $QC_r$ in time to lock it and therefore will not vote for $B_{r+1}$. This allows $L_{r+1}$ to use $TC_r$ to justify making a new proposal with a parent that it knows to be certified.

However, even this measure is not enough to ensure that every honest leader that proposes during a period of synchrony is able to produce a certified proposal. Consider $L_r$, the first such leader to propose in a given sequence of leaders. If $L_r$ proposes a Fallback Recovery Proposal, then this proposal will become certified (see the proof for Lemma 10). Similarly, if all honest processes lock $QC_{r-1}$, then the Chained Moonshot QC Processing Rule ensures that $L_r$ will extend this proposal with $B_{r-1} \Longleftarrow B_r$, which will subsequently become certified. However, because $L_{r-1}$ is Byzantine the adversary can deliver $B_{r-1}$ to the honest processes just before their round timers expire. Variability in network latency all but guarantees that the round timers of the honest processes will differ by some small margin (these bounds are examined more precisely in Section VII), making it possible for the adversary to ensure that at least one honest process receives $QC_{r-1}$ before sending $T_{r-1}$ in order to prevent $TC_{r-1}$ from forming if the Byzantine processes remain silent. Furthermore, it can ensure that the remaining honest processes receive $QC_{r-1}$ after they send $T_{r-1}$, preventing them from locking the QC and thus from voting for $B_{r-1} \Longleftarrow B_r$. This requires very precise timing on the behalf of the adversary, but is possible nevertheless.

Importantly though, in order to execute this attack the adversary necessarily causes $B_{r-1}$ to become certified. In the worst case, only one honest process will have locked $QC_{r-1}$, but all will have observed it and thus will report it in any Timeout messages that they send for $r$ or greater, ensuring that every TC for $r$ or greater will include this QC. Therefore, the inevitable Fallback Recovery Proposal made by $L_{r+1}$ will necessarily extend the proposal of $L_{r-1}$.

Consequently, if Chained Moonshot is implemented with a round-robin leader election function then these guarantees together ensure that every honest process will commit at least $2f+1$ blocks for every $n$ rounds of the protocol, at least $f+1$ of which are guaranteed to be honest. We provide a rigorous proof of this claim in Section VII.

### *E. Round Expiry*

We use the set $E$ to help preserve liveness. As defined in Table I, $E$ tracks all of the rounds that an honest process $v$ considers to have expired but has not yet garbage collected. This prevents the following liveness attack, which would be possible if the algorithm instead only tracked the highest round for which $v$ had sent a Timeout message:

Suppose that $QC_{r-2}$ does not exist and that the adversary causes an honest process to enter a round $r$ via $TC_{r-2}$, the highest round of any honest process, at least $\tau$ before GST begins. This enables it to deliver $f+1$ $T_{r-1}$ to an honest process in some lower round before it sends $T_{r-2}$, which Lemma 7 proves is only guaranteed to occur before $2\Delta$ after GST. Consequently, if we say that our process should not send $T_{r-2}$ because it has already sent $T_{r-1}$, then if the Byzantine processes do not send their $T_{r-2}$ messages to the remaining honest processes and $QC_{r-2}$ does not exist, then the remaining honest processes will never be able to construct $TC_{r-2}$ and will become permanently stuck in $r-2$.

### *F. Timeout Sync*

Per Algorithm 2, an honest process will not trigger the Timeout Sync Rule for $r$ if it has already observed $QC_r$. Without this requirement, the adversary can violate the Safety of the protocol (specifically, Lemma 2 will not hold) by causing two different proposals to become certified for $r+1$ as follows:

Suppose that network is in an asynchronous interval, meaning that the adversary can delay select messages arbitrarily. Suppose that the adversary delivers both $QC_r$ and a valid Normal Proposal containing a $B_{r+1}$ that extends the certified $B_r$, to up to $2f$ honest processes such that they all broadcast Prepare votes for $B_{r+1}$. The adversary can now construct $QC_{r+1}$ and wait for the honest processes still in $r$ to eventually send $T_r$. Subsequently, it can also cause the $f$ Byzantine processes to send $T_r$ messages to the honest processes in $r+2$. Without the current restriction, this would cause these processes to trigger the Timeout Sync Rule and send $T_r$ messages of their own, giving the adversary control of $TC_r$. Therefore, if $L_{r+1}$ is either honest and still in $r+1$, or if it is Byzantine, then the adversary can cause it to propose a Fallback Recovery Proposal containing $B'_{r+1}$. The current Fallback Vote Rule will allow the honest processes in $r+2$ to vote for this proposal since it does not check if the process has already voted for a Normal Proposal for the same round, enabling the adversary to create competing QCs for $r+1$.

### *G. TC Broadcasting*

Although the current Timeout Sync Rule is sufficient under our theoretical model, it might not be suitable for a practical implementation. The upper bound on the network delay, $\Delta$, is difficult to approximate in practice. If this value is set too low in an actual implementation then it is possible that a process will garbage collect its $T_r$ messages before it is able to successfully deliver $T_r$ to some of its honest peers. Should this occur then it is possible that these peers might never be able to construct $TC_r$ and thus the protocol might halt. Consequently, we observe that it might be more practical to use a different mechanism for round synchronisation in the Fallback Path.

A simple alternative would be to have processes broadcast TCs in the same manner that they do QCs. However, this would give Chained Moonshot an overall network-wide communication complexity of $O(n^3)$, since a TC must contain $O(n)$ information in order to prove that $qc'$ was indeed the highest QC submitted by the processes whose Timeout messages were used to construct $TC_r$. Such a high communication complexity may be undesirable though, so we also propose another alternative.

Instead of broadcasting $TC_r$, we speculate that validators should be able to unicast it to $L_{r+1}$ and multicast a threshold signature on $r$ to their remaining peers. Additionally, the Timeout Sync Rule should be modified to cause a validator to send $T_r$ whenever it observes such a threshold signature for $r$ without having already observed $QC_r$. The Liveness proofs given in Section VII rely on honest processes being able to synchronise to the same round within a tight interval after GST.

Having processes multicast the threshold signature on $r$ would actually reduce the current bound from $2\Delta$ to $\Delta$. Likewise, requiring the processes to unicast $TC_r$ to $L_{r+1}$ and the updated Timeout Sync Rule together should ensure that if any process observes a threshold signature on $r$ derived from $T_r$ messages then $L_{r+1}$ will eventually observe $TC_r$. We previously mentioned an optimisation that prevents the adversary from spamming validators with messages for higher rounds by including threshold signatures on $r$ inside Normal Proposal, Prepare and Timeout messages. We observe that if these two optimisations are implemented together then each type of threshold signature on $r$ should also cover a unique domain identifier to distinguish them from one another and thus to preserve the latter conclusion regarding $L_{r+1}$'s guaranteed observation of $TC_r$. We name this optimisation speculative because we have yet to complete formal proofs showing that the proposed changes are sufficient to preserve the properties of the current protocol.

### H. Complexity Analysis

The network-wide asymptotic communication complexity of the Chained Moonshot protocol depends upon the method employed to aggregate Prepare votes into QCs. If a QC is simply a collection containing a quorum of Prepare messages, which we assume are of size $O(1)$, then its size is $O(n)$. In this case, Chained Moonshot's broadcasting of QCs incurs a cost of $O(n^3)$, since all $n$ processes are required to send $O(n)$ sized messages to each of their $n$ peers. However, it is well-known that signatures can be compressed using threshold cryptography. Therefore, if QCs are instead constructed by aggregating a quorum of signature shares into a single quorum threshold signature, then their size reduces to $O(1)$, reducing the cost of QC broadcasting to $O(n^2)$.

Comparatively, threshold cryptography cannot be employed to reduce the size of TCs. As previously mentioned, this is because a TC must prove that its $qc'$ was indeed the highest QC submitted by the processes whose Timeout messages were used to construct the TC in order for the parent of any related Fallback Recovery Proposal to be justified. As an aside, if TCs were not required to include this proof then the current rule for voting on Fallback Recovery Proposals would be unsafe: A Byzantine proposer could initiate a fork by making the parent of its Fallback Recovery Proposal arbitrarily far in the past. Therefore, since TCs must always contain such a proof, their size is necessarily at least $O(n)$. Consequently, the Fallback Recovery Proposal action also incurs a cost of $O(n^2)$. By contrast, the Normal Proposal action has a complexity of only $O(n)$, assuming that blocks are $O(1)$ in size.

We assume that Prepares and Timeouts are also $O(1)$ in size. Thus, since Chained Moonshot requires processes to broadcast Prepare and Timeout messages, the corresponding actions incur a communication complexity of $O(n^2)$.

Therefore, overall, Chained Moonshot exhibits an communication complexity of $O(n^2)$ or $O(n^3)$ per round in both the Normal and Fallback Path, depending on how QCs are constructed. However, we should also consider the worst-case communication cost required for Chained Moonshot to commit a new block. Assume for a moment that our previous claim that if $L$ is a round-robin leader election function then every honest process will commit at least $2f+1$ blocks for every $n$ rounds of the protocol is true. Therefore, by extension, if $L$ is deterministically fair then every honest process will commit at least $k(2f+1)$ blocks for every $kn$ rounds of the protocol. Consequently, since $k(2f+1)$ and $kn$ are both $O(n)$, if $L$ is deterministically fair then Chained Moonshot produces $O(n)$ blocks that will eventually be committed by every honest process, every $O(n)$ rounds. Thus, it still requires only $O(n^2)$ communication in the worst case per decision, assuming $O(1)$ sized QCs.

## VI. Chained Moonshot with Efficient Voting

We previously observed that the communication overhead of the preceding Chained Moonshot protocol can be reduced by including block *digests* in Prepare and QC messages rather than the blocks themselves. Implementing this optimisation requires several modifications to the specification provided in Section IV.

### A. Core Protocol Modifications

Firstly, rather than containing blocks, Prepare and QC messages should instead contain both the digest and round number of the related block. Furthermore, if QCs are also made to include the digest of the parent of the related block then the semantics of the `tryCommit` function can remain unchanged. Otherwise, QCs may omit this field if `tryCommit` is provided with access to certified blocks and is also invoked as each new certified block arrives. Most significantly though, implementing this optimisation requires the addition of a synchronisation subprotocol to ensure that the protocol maintains the *Liveness* SMR property.

### B. Block Synchronisation Protocol

It is possible for processes running Chained Moonshot to fall behind their peers due to either normal network asynchrony or the adversary deliberately delaying messages. Our assumption of perfect communication channels and the requirement that honest leaders broadcast their proposals together ensure that all blocks sent by honest leaders will eventually be delivered to all honest validators. However, perfect channels are insufficient to ensure that every validator eventually receives every finalised block when Prepare messages and QCs do not contain the related block. This is because it is still possible for Byzantine leaders to censor a subset of the honest validators when broadcasting their Proposals. Accordingly, we now present a simple Block Synchronisation Protocol that ensures that if one honest process commits a block then all others will eventually do the same.

A process $v$ initiates the Block Synchronisation Protocol when it becomes aware of the existence of a certified block that it has yet to receive. This first occurs when $v$ processes a QC for a block $B$ that it does not have, in which case it multicasts a *Sync Request* message containing the digest of $B$ to $2f+1$ of its peers. Any honest peer of $v$ that receives such a Sync Request first ensures that the sender has not exceeded the agreed upon rate-limit for Sync Requests before checking its storage for the requested block. If it has the block the server then unicasts a *Sync Response* containing $B$ to the sender. The existence of the QC for $B$ implies that at least $f+1$ honest processes received $B$, so since $n = 3f+1$, $v$ is guaranteed to eventually receive $B$ from at least one honest peer. After receiving $B$, $v$ authenticates the Sync Response by confirming that it had previously requested $B$ and then processes $B$. If $v$ discovers that it is missing the parent of $B$ while processing $B$ then it sends another Sync Request, repeating the prior steps until it has processed all blocks between its last committed block and $B$. Importantly, the safety proofs given in Section VII show that $B$ is guaranteed to be a descendent of $v$'s last committed block as long as the Byzantine threshold remains intact, so this protocol is guaranteed to eventually terminate.

### C. Optimisations and Analysis

We observe that the $2f+1$ multicast of the Sync Request message can be reduced to an $f+1$ multicast when the sender has the related QC in its possession, if the QC identifies it contributors. This is because the Chained Moonshot Voting Rule ensures that each honest contributor is guaranteed to possess the related block.

The presented Block Synchronisation Protocol has a best-case latency of $2\delta$ after GST and communication complexity of $\Theta(f+1)$ messages per block, assuming a QC implementation that preserves the identities of the voters. However, it is possible to achieve a communication complexity of $\Omega(1)$, $O(f+1)$ per block in the same setting by having $v$ contact its peers one at a time. In this case, the latency remains $2\delta$ in the best case but increases to $(f+1)\Delta$ in the worst case.

The communication complexity of the full synchronisation process can be further optimised by having $v$ request multiple blocks from the same process once it has identified an honest server. In this variant, $v$ follows the previously described protocol for the first missing block. After receiving this block from $v_s$, $v$ then directs all future Sync Requests to $v_s$ until it fails to receive a response within some predetermined timeout interval, in which case it repeats the original protocol until it receives the requested block from $s''$, and so on until it has synchronised all of its missing blocks.

The introduction of this subprotocol to Chained Moonshot requires the network to keep a record of processed blocks so that they can be served to desynchronised or new validators when necessary. We observe that the long-term costs of this requirement can be mitigated by offloading the responsibility of maintaining the full history to external *archive nodes*, allowing the validators to maintain only the most recent history.

## VII. Correctness Proofs

We now present correctness proofs for Chained Moonshot and show that our protocol satisfies the Safety and Liveness properties of SMR from Definition 3. These proofs cover both the basic protocol presented in Section IV and the variant with efficient voting discussed in Section VI.

We first recollect the rules of Chained Moonshot in Tables III and IV before establishing some new definitions to aid us in the proofs.

**Definition 5** (Canonical Block)**.** *$B_r$ is canonical iff every certified block $B_{r'}$ with $r' \geq r$ has $B_r \Longleftarrow^* B_{r'}$.*

**Definition 6** (Local Direct-Commit)**.** *$B_r$ is locally direct-committed (LDC) by a process $v$ when $v$ executes the Two-Chain Commit Rule on $B_r$.*

**Definition 7** (Local Commit)**.** *$B_r$ is locally committed (LC) by $v$ when $v$ LDCs $B_{r'}$ such that $B_r \Longleftarrow^* B_{r'}$.*

**Definition 8** (Honest Majority Lock)**.** *$B_r$ is honest-majority locked (HML) iff there are at least $f+1$ honest processes that lock the QC for $B_r$.*

**Definition 9** (Universal Lock)**.** *$B_r$ is universally locked (UL) iff all honest processes lock the QC for $B_r$.*

TABLE III
RULES OF CHAINED MOONSHOT NORMAL PATH FOR $v \in \mathcal{V}$

| |
|---|
| **Garbage Collection.** A process $v$ garbage collects Prepare, Proposal and Timeout messages for $r' < r$ per the semantics of the $cleanup$ function upon entering $r+1$. It likewise garbage collects QCs and blocks for $r' \leq r$ per the semantics of the `commit` function upon appending $B_r$ to $\mathbf{B_v}$. |
| **Lock.** A process $v$ locks a block $B_r$, meaning it sets $qc_l$ to the QC for $B_r$ upon receiving this QC, only if it is in round $r+1$ or lower, it has not sent $T_r$ and $r > qc_l.B.r$. |
| **Proposal: Normal.** Whilst in $r$, $v$ broadcasts a Normal Proposal $N(B_r)$ that extends the first Normal Proposal $N(B_{r-1})$ that it receives from $L_{r-1}$, if it is $L_r$. |
| **Proposal: QC Extension.** If $v$ observes $QC_{r-1}$ whilst in round $r' \leq r$, is $L_r$ and has not yet created either a Fallback Recovery Proposal for $r$ or a Normal Proposal for $r$ that references the block certified by $QC_{r-1}$ as its parent, then it creates a Normal Proposal extending the block certified by this QC. |
| **QC Sync.** Upon observing a QC for a given round for the first time, $v$ broadcasts it. |
| **Round Transition: Normal.** $v$ enters $r+1$ from $r' < r+1$ after observing a QC for $r-1$. |
| **Two-Chain Commit.** Upon receiving QCs for both $B_r$ and its child block $B_{r'}$ such that $r' = r+1$, $v$ schedules $B_r$ and its ancestors for commit per the semantics of the `commit` function defined in Table II. |
| **Vote Broadcast.** $v$ broadcasts all votes. |
| **Vote: Normal.** Whilst in $r+1$ a process $v$ sends a Prepare vote for a Normal Proposal $N(B_r)$ received from $L_r$ with $B_{r-1} \Longleftarrow B_r$, only if it has neither sent $T_r$ nor already voted for a Normal Proposal for $r$, and it is locked on the QC for $B_{r-1}$. |

TABLE IV
RULES OF CHAINED MOONSHOT FALLBACK PATH FOR $v \in \mathcal{V}$

| |
|---|
| **Proposal: Fallback.** Upon observing $TC_{r-1}$ whilst in $r' \leq r$, $v$ creates a Fallback Recovery Proposal $F(B_r)$ with $B'' \Longleftarrow B_r$, where $B''$ is the block certified by the QC with the highest round number included in $TC_{r-1}$, only if it is $L_r$. |
| **Proposal: Fallback Extension.** If $v$ receives a Fallback Recovery Proposal $F(B_{r-1})$ from $L_{r-1}$ with $B'' \Longleftarrow B_{r-1}$ whilst in round $r$, then it creates a Normal Proposal $N(B_r)$ with $B_{r-1} \Longleftarrow B_r$ only if $F$ is justified by a well-formed $TC_{r-1}$, $B''$ is the block certified by the QC with the maximum round number included in $F$, $v$ is $L_r$ and $F$ is the first Fallback Recovery Proposal for $r-1$ that satisfies these conditions. |
| **Round Transition: Fallback.** $v$ enters $r+1$ from $r' < r+1$ after observing a TC for $r-1$. |
| **Timeout.** $v$ resets its round timer upon entering round $r$ and broadcasts $T_{r-1}$ if it remains in $r$ for $\tau$ where $\tau > 4\Delta$, without otherwise broadcasting this message. |
| **Timeout Sync.** If $v$ observes $f+1$ Timeout messages for $r$ before garbage collecting the $T_r$ messages that it has received and while having yet to append a block for $r$ or higher to $\mathbf{B_v}$ and not having $QC_r$ in $U$, then it broadcasts $T_r$ if it has not already done so. Likewise, upon observing $TC_r$ and having not yet sent $T_r$, $v$ broadcasts $T_r$. |
| **Vote: Fallback.** Whilst in $r+1$ $v$ sends a Prepare vote for a block $B_r$ with $B' \Longleftarrow B_r$ received in a Fallback Recovery Proposal $F$ from $L_r$, only if it has neither sent $T_r$ nor already voted for a Fallback Recovery Proposal for $r$, $F$ is justified by $TC_{r-1}$ and $B'$ is the block certified by the QC with the maximum round number included in $TC_{r-1}$. |

### A. Safety

**Lemma 1** (No Timeout Before Lock)**.** *If an honest process $v$ locks $QC_r$ then it must not have sent $T_r$ before doing so.*

*Proof:* Suppose that $v$ locks $QC_r$ after sending $T_r$. Therefore, by the Timeout Rule, $v$ must have added the round identifier of $r$ to $E$ upon sending $T_r$. Moreover, by the Garbage Collection Rule, $v$ cannot remove $r$ from $E$ until it has spent at least $2\Delta$ in $r+2$. Consequently, since the Lock Rule requires $v$ to be in $r+1$ or lower in order for it to lock $QC_r$, $v$ must have had $r$ in $E$ when attempted to lock $QC_r$. However, the Lock Rule also requires that $v$ must not have $r$ in $E$, meaning that it would have failed to lock $QC_r$, contradicting the assumption that it did so. ■

We use the term *Normal $QC_r$* in the informal name of Lemma 2 to refer to a QC for a Normal Proposal for round $r$.

**Lemma 2** ($TC_{r-1}$ implies no Normal $QC_r$)**.** *If $TC_{r-1}$ exists then no Normal Proposal for $r$ will ever become certified.*

*Proof:* Suppose that $TC_{r-1}$ exists and $QC_r$ certifies a Normal Proposal containing the block $B_r$. By the Normal Vote Rule, the existence of $QC_r$ implies that a group of at least $f+1$ honest processes, say $H_1$, must have locked the parent of $B_r$, which in turn must have been proposed for $r-1$. Thus, by the Lock Rule, these processes must have observed $QC_{r-1}$ whilst in round $r$ or lower. Moreover, since $TC_{r-1}$ exists, at least $f+1$ honest processes must have sent $T_{r-1}$ messages. Let $H_2$ contain the first $f+1$ honest processes to send $T_{r-1}$. By quorum intersection, $H_1$ and $H_2$ must have at least one member in common. Furthermore, by Lemma 1, none of the processes in $H_1$ can have sent $T_{r-1}$ before locking $QC_{r-1}$. Therefore, at least one honest process, say $v$, must have sent $T_{r-1}$ after locking $QC_{r-1}$. However, since the Normal Round Transition Rule would have caused $v$ to enter $r+1$ when it observed $QC_{r-1}$ if it had not already entered a higher round, $v$ must have sent $T_{r-1}$ after entering $r+1$ or higher. Consequently, it cannot have sent $T_{r-1}$ as a result of the Timeout Rule and thus must have done so due to the Timeout Sync Rule. Moreover, because $v$ is a member of $H_2$, it cannot have sent $T_{r-1}$ after observing $TC_{r-1}$, because $TC_{r-1}$ cannot exist until the members of $H_2$ have sent their $T_{r-1}$ messages. Therefore $v$ must have sent $T_{r-1}$ after observing $f+1$ $T_{r-1}$ messages without having either appended a block for $r' \geq r-1$ to its local blockchain or having $QC_{r-1}$ in $U$. However, recall that a process is required to add $QC_{r-1}$ to $U$ upon observing it for the first time, and that it may only remove it from $U$ upon appending a block for $r' \geq r-1$ to its local blockchain. Hence, since we have concluded that when $v$ sent $T_{r-1}$ it must have both already observed $QC_{r-1}$ and not appended a block for $r' \geq r-1$ to its local blockchain, $v$ must have had $QC_r$ in $U$ when it broadcasted $T_{r-1}$. However, this violates the Timeout Sync Rule and thus contradicts the definition of $v$ as being honest. ■

**Lemma 3** (Round Safety)**.** *Suppose two processes $v_i$ and $v_j$ observe QCs for blocks $B_i$ and $B_j$, respectively. If $B_i.r = B_j.r$ then $B_i = B_j$.*

*Proof:* Suppose $B_i.r = B_j.r$ and $B_i \neq B_j$. By the Vote Rule and the requirement that QCs be derived from a quorum of valid Prepare messages for the same block, the existence of the QCs for $B_i$ and $B_j$ implies that at least $f+1$ honest processes voted for each of them in $B_i.r+1$. Furthermore, since there are only $2f+1$ honest processes, at least one of these processes must have voted for both $B_i$ and $B_j$. However, if $B_i$ and $B_j$ were both proposed as Normal Proposals or both as Fallback Recovery Proposals then because the rules for voting only allow an honest process to vote for one proposal of each type per round, no honest process could have voted for both blocks. Alternatively, if $B_i$ were proposed in a Normal Proposal $N$ and $B_j$ in a Fallback Recovery Proposal $F$, or vice-versa, then the Vote Rule allows honest processes to vote for both blocks. However, since the well-formedness rule for Fallback Recovery Proposals requires that $F$ be justified by $TC_{r-1}$, by Lemma 2, $N.B$ will never be certified. Therefore, $B_i = B_j$. ■

**Lemma 4** (Sequential Progress)**.** *If an honest process $v$ enters round $r$ then at least one honest process must have already entered $r-1$.*

*Proof:* The Round Transition Rules require $v$ to observe either $QC_{r-2}$ or $TC_{r-2}$ in order to enter $r$. Furthermore, at least $f+1$ honest processes must vote towards each of these certificates. In the case of $QC_{r-2}$, the Normal and Fallback Vote Rules require these processes to enter $r-1$ before they may do so. In the case of $TC_{r-2}$, the Timeout and Timeout Sync Rules together imply that at least one honest process must enter $r-1$ before any honest process can send $T_{r-2}$. Thus, in either case, $v$ can only enter $r$ if at least one honest process has already entered $r-1$. ■

**Lemma 5** (Non-Decreasing Max QC)**.** *If an honest process $v$ adds $QC_r$ to $U$ then every Timeout message that it sends after doing so contains a QC for a round $r' \geq r$.*

*Proof:* Recall that the Garbage Collection Rule only allows $v$ to remove $QC_r$ from $U$ upon appending a block for $r'' \geq r$ to $\mathbf{B_v}$. Therefore, by the Two-Chain Commit Rule, $v$ must observe $QC_s$ and $QC_{s+1}$ such that $s \geq r$ in order to remove $QC_r$ from $U$. Thus, $v$ would have added both $QC_s$ and $QC_{s+1}$ to $U$ before removing $QC_r$ from $U$. Consequently, since $s \geq r$, every subsequent invocation of `maxQC(`$U$`)` will return a QC for $r$ or higher. Therefore, since the Timeout Rule requires $v$ to include `maxQC(`$U$`)` in every Timeout message that it sends, every Timeout message sent by $v$ after adding $QC_r$ to $U$ will contain a QC for $r$ or higher. ■

**Lemma 6** (LDC is Unique)**.** *If an honest process might LDC $B_r$ then for every certified block $B_{r'}$ such that $r' \geq r$, $B_r \Longleftarrow^* B_{r'}$.*

*Proof:* We prove this claim by induction on the round number. Lemma 3 proves that if $r' = r$ then $B_{r'} = B_r$ and thus $B_r \Longleftarrow^* B_{r'}$. Consider the case when $r' > r$:

***Base Case****:* $r' = r+1$. Once again, Lemma 3 proves that only one block can become certified in a given round. Therefore, because $B_{r'}$ is certified for $r+1$ and $B_r$ may be LDC, it follows from Definition 6 and the Two-Chain Commit Rule that $B_r \Longleftarrow^* B_{r'}$.

***Inductive Step****:* We assume that the lemma holds up to round $k$ such that $k > r$ and complete the proof for $r' = k+1$. Therefore, if $B_k \Longleftarrow B_{r'}$ then $B_r \Longleftarrow^* B_{r'}$, so the only case that remains is when $B_{r''} \Longleftarrow B_{r'}$ such that $r'' < r$. Recall that the Normal Vote Rule only allows honest processes to vote for a Normal Proposal if its parent was proposed in the previous round. Therefore, because $B_{r'}$ is certified and since $r'' < r < r'-1$, $B_{r'}$ must have been proposed as a Fallback Recovery Proposal. We now show that the TC justifying this Fallback Recovery Proposal will necessarily contain $QC_r$ or higher, contradicting the requirement that $r'' < r$.

By Definition 6 and the Two-Chain Commit Rule, an honest process will not LDC $B_r$ unless it observes a QC for $B_{r+1}$ such that $B_r \Longleftarrow B_{r+1}$. Therefore, since an honest process might LDC $B_r$, $QC_{r+1}$ must exist. Consider the type of $B_{r+1}$.

If $B_{r+1}$ were a Normal Proposal then, because it is certified and thus at least $f+1$ honest processes must have voted for it, by the Normal Vote Rule, these processes must have locked $QC_r$. Let $H$ represent the first $f+1$ honest processes to vote for $B_{r+1}$. Therefore, by the Lock Rule, $H$ must have observed $QC_r$ whilst in round $r+1$ or lower and hence cannot have broadcasted $T_{r+1}$ as a result of the Timeout Rule before they did so, which only allows a process to do so whilst in $r+2$. Therefore, if any process in $H$ sent $T_{r+1}$ before observing $QC_r$, it must have done so due to the Timeout Sync Rule. However, by the Normal Vote Rule, $H$ can only have voted for $B_{r+1}$ if they did not have the round identifier of $r+1$ in $E$. Therefore, since a process is required to add the identifier of $r+1$ to $E$ when it sends $T_{r+1}$, and because the Garbage Collection Rule only allows it to remove this identifier from $E$ after having spent at least $2\Delta$ in $r+3$, because these processes must have voted for $B_{r+1}$ whilst in $r+2$, none of them can have sent $T_{r+1}$ before observing $QC_r$.

Alternatively, if $B_{r+1}$ were a Fallback Recovery Proposal then, by the Fallback Vote Rule, the $TC_r$ justifying this proposal must have contained $QC_r$ as its highest QC. Therefore, since $H$ must have voted for $B_{r+1}$ in order for it to be certified, by the Fallback Vote Rule, every process in $H$ must have done so whilst in $r+2$ and without having the round identifier of $r+1$ in $E$. Therefore every process in $H$ must have observed $QC_r$ whilst in $r+2$ or lower. Thus, as reasoned in the former case, they all must also have observed $QC_r$ before sending $T_{r+1}$.

In either case, $H$ must have observed $QC_r$ before sending $T_{r+1}$. Thus, by Lemma 5, any Timeout message for $r+1$ sent by these $f+1$ honest processes will necessarily contain a QC for round $r$ or higher. Therefore, because there are only $2f+1$ honest processes, $n = 3f+1$ and since TCs must be constructed from at least $2f+1$ Timeout messages, every TC for $r+1$ will contain a QC for round $r$ or higher. Therefore, since $r' > r+1$ and because $B_{r'}$ is necessarily a Fallback Recovery Proposal, if $B_{r'} = r+2$ then $r'' \geq r$, contradicting the earlier conclusion that $r'' < r$.

Moreover, if every Timeout message sent by $H$ for every round greater than $r+1$ is also guaranteed to contain a QC for $r$ or higher, then we can extend this conclusion to every possible value of $r'$. Suppose, then, that some honest process $v \in H$ sent a Timeout message for $r^* > r+1$ containing a QC for a round lower than $r$. From our earlier conclusions, $v$ must have sent this message before observing $QC_r$ and whilst in $r+2$ or lower. Therefore, $v$ cannot have sent $T_{r^*}$ as a result of the Timeout Rule and thus must have done so due to the Timeout Sync Rule. Consequently, since this implies that at least $f+1$ $T_{r^*}$ messages must already have existed when $v$ sent $T_{r^*}$, the Timeout and Timeout Sync Rules together imply that at least one honest process, say $v'$, must have spent $\tau$ in $r^*+1$ before this time. Therefore, $v'$ must have entered $r^*+1$ whilst all of the processes in $H$ were still in $r+2$ or lower and before they sent either Prepare messages for $B_{r+1}$ or $T_{r+1}$. Moreover, since $B_{r+1}$ must be certified, by Lemma 3, no $QC_{r+1}$ can exist before $H$ vote for $B_{r+1}$. Therefore, no honest process can have entered $r+3$ via the Normal Round Transition Rule before $H$ voted for $B_{r+1}$. Likewise, neither can any honest process have entered $r+3$ via the Fallback Round Transition Rule before at least one honest process in $H$ sent $T_{r+1}$, since at most $2f$ $T_{r+1}$ messages can exist before this time. Consequently, since no honest process can have entered $r+3$ until one of these two events occurs, by Lemma 4, neither can any honest process have entered a round after $r+3$ before this time. However, this contradicts our earlier conclusion that $v'$ must have entered $r^*+1 \geq r+3$ before both of these events.

Therefore, $B_r \Longleftarrow^* B_{r'}$ for every certified block $B_{r'}$ such that $r' \geq r$. ■

Corollary 1 follows from Lemma 6 and the fact that every LDC block is necessarily certified.

**Corollary 1** (Consistency). *If $B_r$ and $B_{r'}$ are both LDC then either $B_r \Longleftarrow^* B_{r'}$ or $B_{r'} \Longleftarrow^* B_r$.*

**Theorem 1** (Safety). *For every run $R \in \mathcal{R}_\mathcal{P}$, for each pair of honest processes $(v_i, v_j) \in \mathcal{V} \times \mathcal{V}$, at each moment during $R$ either $\mathbf{B_{v_i}} \preceq \mathbf{B_{v_j}}$ or $\mathbf{B_{v_j}} \preceq \mathbf{B_{v_i}}$.*

*Proof:* Let $\mathbf{B}_{v_i} = B_0^1 \Longleftarrow^* B_s^1$ and $\mathbf{B}_{v_j} = B_0^2 \Longleftarrow^* B_t^2$. Then for every height $0 \leq l \leq s$, we have that $v_i$ LC $B_l^1$ and for every height $0 \leq h \leq t$, $v_j$ LC $B_h^2$. Therefore, from Definition 7, we have that $B_s^1$ and $B_t^2$ are LDC. Therefore, from Corollary 1 we have that $B_s^1 \Longleftarrow^* B_t^2$ or $B_t^2 \Longleftarrow^* B_s^1$. Thus, either $\mathbf{B}_{v_i} \preceq \mathbf{B}_{v_j}$ or $\mathbf{B}_{v_j} \preceq \mathbf{B}_{v_i}$ at every moment of every $R \in \mathcal{R}_\mathcal{P}$. ■

### B. Liveness

We recall that our assumption of perfect communication channels implies that all messages between the processes in $\mathcal{V}$ are eventually delivered. Moreover, since we also assume that these channels are *partially synchronous*, the upper bound on this delivery during each synchronous period during the protocol run is $\Delta$. For the sake of the following proofs, we reason in the context of one such synchronous period, the beginning of which we denote by either GST or $t_g$. We carry these assumptions and definitions forward in the following proofs.

We begin by showing that all honest processes are guaranteed to continue to enter new rounds after GST.

**Lemma 7** (Round Sync)**.** *Let $r$ be the highest round of any honest process at time $t \geq t_g$. All honest processes enter $r$ or greater before $t + 2\Delta$.*

*Proof:* Let $v$ be an honest process in $r$ at $t$. Recall that the Round Transition Rules allow a process to enter round $r$ only if it observes $QC_{r-2}$ or $TC_{r-2}$. In the former case, the QC Sync Rule ensures that $v$ would have broadcasted $QC_{r-2}$ just before it entered $r$. Therefore, all honest processes will observe this certificate and enter $r$ via the Normal Round Transition Rule before $t + \Delta$, if they do not enter a higher round first.

In the latter case, the existence of $TC_{r-2}$ implies that at least $f + 1$ honest processes must have broadcasted $T_{r-2}$ before $v$ entered $r$ and thus before $t$. If all honest processes broadcast $T_{r-2}$ before $t + \Delta$ then, since there are $2f + 1$ honest processes and because TCs require $2f + 1$ Timeout votes to construct, all processes will be able to construct $TC_{r-2}$ before $t + 2\Delta$ and thus will enter $r$ by the Fallback Round Transition Rule by this time.

Suppose, then, that some honest process $v'$ does not broadcast $T_{r-2}$ before $t + \Delta$. However, $v'$ is guaranteed to observe the aforementioned $f + 1$ $T_{r-2}$ messages before this time. Therefore, by the Timeout Sync Rule, $v'$ must have $QC_{r-2}$ in $U$ upon observing these messages, or it must either have appended a block for $r - 2$ or higher to $\mathbf{B_{v'}}$, or garbage collected its $T_{r-2}$ messages before this time. In the first case, since $v'$ can only have added $QC_{r-2}$ to $U$ after observing it, it would also have broadcasted it, so the proof for this case has already been covered. Likewise, in the second case, by the Two-Chain Commit Rule, $v'$ must have observed QCs for two consecutive rounds greater than $r - 2$ in order to append $B_{r-2}$ or higher to its local blockchain, so this case has also been covered.

Suppose, then, that $v'$ had garbage collected the $T_{r-2}$ messages that it had received before $t + \Delta$. Therefore, by the Garbage Collection Rule, $v'$ must have spent at least $2\Delta$ in some round $r' \geq r$ before $t + \Delta$. However, this implies that $v'$ entered $r'$ no later than $t - \Delta$, contradicting the definition of $r$ as being the highest round of any honest process at $t$ if $r' > r$. Therefore, $r' = r$. Furthermore, $v'$ cannot have entered $r$ via $TC_{r-2}$, otherwise the Timeout Sync Rule would have caused it to broadcast $T_{r-2}$, contradicting the assumption that it does not do so. Therefore, $v'$ must have entered $r$ via $QC_{r-2}$, which the QC Sync Rule once again ensures all honest processes will observe before $t + \Delta$. ■

**Lemma 8** (Certificate Progress)**.** *Let $r$ be the highest round of any honest process at time $t \geq t_g$. If $\tau > 4\Delta$ then all honest processes observe a certificate for $r' \geq r - 1$ before $t + 4\Delta + \tau$.*

*Proof:* Suppose that some honest process fails to observe a certificate for $r' \geq r - 1$ before $t + 4\Delta + \tau$. Therefore, by the QC Sync Rule, no honest process may observe a QC for $r - 1$ or greater before $t + 3\Delta + \tau$. Furthermore, if all honest processes broadcast $T_{r-1}$ before $t + 3\Delta + \tau$ then all processes will be able to construct $TC_{r-1}$ before $t{+}4\Delta{+}\tau$, even if the Byzantine ones remain silent.

Suppose, then, that some honest process $v'$ fails to broadcast $T_{r-1}$ before $t + 3\Delta + \tau$. Lemma 7 shows that $v'$ will enter $r$ or higher before $t + 2\Delta$. Therefore, the Timeout Rule ensures that if $v'$ remains in $r$ until $t + 2\Delta + \tau$ then it will have broadcasted $T_{r-1}$ by this time. However, since $v'$ must not send $T_{r-1}$ before $t{+}3\Delta{+}\tau$ and because no honest process can observe a QC for $r - 1$ or greater within the same interval, $v'$ must enter $r_h > r$ via $TC_{r_h-2}$ before $t + 2\Delta + \tau$. However, if $r_h = r{+}1$ then $v'$ would have observed $TC_{r-1}$ and the Timeout Sync Rule would have caused it to send $T_{r-1}$, contradicting the earlier conclusion that it must not do so before $t + 3\Delta + \tau$. Therefore, $r_h > r + 1$.

Consequently, by Lemma 4 and because no honest process may observe $QC_{r-1}$ or greater before $t{+}3\Delta{+}\tau$, at least one honest process, say $v$, must have entered $r{+}1$ via $TC_{r-1}$ before $v'$ entered $r_h$. More precisely, since $v'$ must enter $r_h$ via $TC_{r_h-2} \geq TC_r$ before $t{+}2\Delta{+}\tau$ and because the Timeout and Timeout Sync Rules together ensure that $TC_r$ cannot exist until at least one honest process has spent at least $\tau$ in $r + 1$, $v$ must have done so before $t + 2\Delta$. Likewise, because $r$ is defined as the greatest round of any honest process at $t$, $v$ cannot have entered $r + 1$ before $t$. Thus, $v$ must enter $r + 1$ at $t_v$ such that $t < t_v < t + 2\Delta$.

Therefore, by Lemma 7, $v'$ will enter $r + 1$ or higher before $t_v + 2\Delta < t + 4\Delta$. However, if $\tau \geq 4\Delta$ then because no honest process can have observed a QC for $r{-}1$ or greater before $t{+}3\Delta{+}\tau$ and neither can any such process have entered $r{+}1$ before $t$, no honest process can have spent $\tau$ in $r{+}1$ before $t_v{+}2\Delta < t{+}4\Delta$. Thus, $TC_r$ cannot exist before this time and, by extension, neither can any TC for any higher round. Consequently, since $v'$ must enter $r{+}1$ or higher before this time, it must do so via $TC_{r-1}$, contradicting our earlier conclusion that it must not observe this TC before $t + 3\Delta + \tau$. ∎

From Lemma 8, we have the following corollary.

**Corollary 2** (Round Progress)**.** *If $M > 4\Delta + \tau$ then for every run $R \in \mathcal{R}_\mathcal{P}$, for each synchronous interval $S \in C_R(M)$, all honest processes continue to enter increasing rounds during $S$.*

We will assume that $M > 4\Delta + \tau$ until we reach Theorem 2. We now show that if an honest leader proposes a Fallback Recovery Proposal after GST then this proposal becomes canonical. For the following Lemmas, let $t_i$ denote the time that the first honest process enters round $r + i$.

**Lemma 9** (Honest Proposals Arrive Before Timeout)**.** *Let $L_r$ be honest and suppose that GST had passed before the first honest process entered $r$. If $L_r$ proposes and $\tau > 3\Delta$ then all of its proposals reach all honest processes before $t_1 + 3\Delta$ and before any of them send $T_r$.*

*Proof:* By the Proposal Rules, $L_r$ is only allowed to propose whilst in $r$. Therefore, since Lemma 7 shows that $L_r$ will enter $r + 1$ or higher before $t_1 + 2\Delta$, if it proposes then it must do so before this time. Consequently, because $L_r$ is honest and so will broadcast its proposal, if it proposes then all processes are guaranteed to observe its proposal before $t_1 + 3\Delta$. Furthermore, since the Timeout and Timeout Sync Rules together imply that at least one honest process must have spent at least $\tau$ in $r + 1$ before any honest process can send $T_r$, if $\tau > 3\Delta$ then no honest process can have broadcasted $T_r$ before receiving $L_r$'s proposal. ∎

**Lemma 10** (Fallback Proposals Are Certified)**.** *Let $L_r$ be an honest leader and suppose that GST had passed before the first honest process entered $r$. If $L_r$ broadcasts a Fallback Recovery Proposal and $\tau > 3\Delta$ then all honest processes observe a QC for this proposal before $t_1 + 4\Delta$.*

*Proof:* $L_r$, being honest, would have sent the same well-formed Fallback Recovery Proposal $F$ to all honest processes, and would only have created one such proposal. Recall that aside from its well-formedness, the validity of a Fallback Recovery Proposal relies only on the state of the $a_f$, $r_c$ and $E$ variables of its recipient. Also recall that Lemma 9 proves that if $\tau > 3\Delta$ then all honest processes will receive $F$ before they send $T_r$ and thus cannot have the round identifier of $r$ in $E$ when they do so. Moreover, nor will any honest process have been able to enter $r+2$ via $TC_r$ before this time. Furthermore, by the premise, neither can any honest process have entered $r+2$ before this time by observing a $QC_r$ for $F$, otherwise the proof would be complete. Moreover, since $F$ is necessarily justified by $TC_{r-1}$, Lemma 2 proves that neither can any honest process have entered $r + 2$ by observing a $QC_r$ for a Normal Proposal made by $L_r$. Thus, by Lemma 4, no honest process can have entered $r'' > r + 1$ before receiving $F$. Consequently, every honest process will have $r_c \leq r + 1$ and $a_f < r$ upon its arrival and will enter $r + 1$, if they have not already done so, upon processing $TC_{r-1}$. Therefore, if $\tau > 3\Delta$ then by Lemma 9 and the Fallback Vote Rule, all honest processes will vote for $F$ before $t_1 + 3\Delta$, so all processes will observe a QC for this proposal before $t_1 + 4\Delta$. ∎

**Lemma 11** (Honest Fallback Proposals Are Canonical)**.** *Let $L_r$ be an honest leader and suppose that GST had passed before the first honest process entered $r$. If $L_r$ proposes $B_r$ as a Fallback Recovery Proposal and $\tau > 4\Delta$ then either $QC_r$ will be UL before $t_1 + 4\Delta$ or every honest process will LDC a block $B_{r'}$ with $B_r \Longleftarrow^* B_{r'}$ where $r' \geq r$ before $t_1 + 5\Delta$.*

*Proof:* By Lemma 10, all honest processes will observe a QC for $L_r$'s Fallback Recovery Proposal before $t_1 + 4\Delta$. Moreover, if $\tau > 4\Delta$ then because $t_1$ is defined as the time that the first honest process entered $r + 1$, no honest process can have sent $T_r$ before observing $QC_r$ and thus no honest process can have the round identifier of $r$ in $E$.

Suppose that all honest processes were in $r + 1$ or lower when they observed $QC_r$. Therefore, the Normal Round Transition Rule implies that they cannot have locked a QC for a higher round before this time. Thus, in this case, by the Lock Rule, all honest processes will lock $QC_r$ thus making it, by Definition 9, UL.

Suppose, then, that some honest process $v'$ observed $QC_r$ after entering $r_h > r + 1$ before $t_1 + 4\Delta$. Therefore, since we have already concluded that $TC_r$ cannot exist before this time, Lemma 4 implies that neither can a TC for any higher round. Thus, $v'$ must have entered $r_h$ via $QC_{r_h-2}$. However, if $r_h = r + 2$ then $v'$ must have entered $r_h$ via $QC_r$, contradicting the assumption that it observed this QC after entering $r_h$. Thus, $r_h > r + 2$. Therefore, by Lemma 4, at least one honest process must have already entered $r + 3$ and, because $TC_{r+1}$ cannot exist by this time, it must have done so via $QC_{r+1}$. Moreover, since $TC_r$ cannot exist by this time, $QC_{r+1}$ cannot certify a Fallback Recovery Proposal and so must certify a Normal Proposal. Therefore, by

Definition 6 and the Two Chain Commit Rule, $B_r$ satisfies the conditions required for LDC. Consequently, by Corollary 1, every LDC block for $r' > r$ will have $B_r \Longleftarrow^* B_{r'}$. Furthermore, since at least $f+1$ honest processes must have locked $QC_r$ for $QC_{r+1}$ to exist, the QC Sync Rule guarantees that all honest processes will observe both $QC_r$ and $QC_{r+1}$ before $t_1+5\Delta$. Therefore, by the Two-Chain Commit Rule, every honest process will LDC a block $B_{r'}$ with $r' \geq r$ and $B_r \Longleftarrow^* B_{r'}$, before this time.

Thus, if $L_r$ is honest and proposes a Fallback Recovery Proposal then either $B_r$ will be UL before $t_1+4\Delta$ or every honest process will LDC a block $B_{r'}$ with $B_r \Longleftarrow^* B_{r'}$ where $r' \geq r$ before $t_1+5\Delta$. ■

**Lemma 12** (Honest Leaders Propose)**.** *If the first honest process to enter $r$ does so after GST, $L_r$ is honest and $\tau > 2\Delta$, then $L_r$ proposes.*

*Proof:* Suppose that $L_r$ does not propose. Therefore, $QC_r$ will never exist so no honest process can ever enter $r+2$ via this certificate. However, recall that we know from Corollary 2 that all honest processes continue to enter increasing rounds after GST. Therefore, by Lemma 4, at least one honest process must eventually enter $r+2$ via $TC_r$. However, the Timeout and Timeout Sync Rules together ensure that $TC_r$ cannot exist until at least one honest process has spent $\tau$ in $r+1$. Let $v$ be the first honest process to send $T_r$ and let the time that $v$ enters $r+1$ be denoted $t_v$. Therefore, since $QC_r$ cannot exist at all and because $TC_r$ cannot exist before $t_v+\tau$, Lemma 4 implies that no honest process can have entered $r' > r+1$ before $t_v+\tau$. Therefore, since Lemma 7 proves that all honest processes including $L_r$ will enter $r+1$ no later than $t_v+2\Delta$, if $\tau > 2\Delta$ and $L_r$ does not propose then $L_r$ must enter $r+1$. However, if $L_r$ enters $r+1$ via $QC_{r-1}$ then the QC Extension Rule ensures that it will create a Normal Proposal extending the block certified by $QC_{r-1}$, contradicting the assumption that it does not propose. Similarly, if $L_r$ enters $r+1$ via $TC_{r-1}$, then it will instead create a Fallback Recovery proposal extending the block certified by the QC with the greatest round included in $TC_{r-1}$, once again contradicting the initial assumption. Therefore, if $\tau > 2\Delta$, the first honest process to enter $r$ does so after GST and $L_r$ is honest, then $L_r$ proposes. ■

**Lemma 13** (No $QC_r$ implies $\mathtt{F}_{r+1}$)**.** *If no honest process enters $r+2$ via $QC_r$, $\tau > 4\Delta$ and $L_{r+1}$ is honest, then $L_{r+1}$ will create a Fallback Recovery Proposal.*

*Proof:* Suppose that $L_{r+1}$ is honest, no honest process enters $r+2$ via $QC_r$ and $L_{r+1}$ does not create a Fallback Recovery Proposal. Therefore, by the Fallback Proposal Rule, $L_{r+1}$ must not enter $r+2$ via $TC_r$. However, because no honest process may enter $r+2$ via $QC_r$, Corollary 2 and Lemma 4 show that at least one honest process must eventually enter $r+2$ via $TC_r$. Therefore, by Lemma 2, the existence of $TC_r$ implies that $QC_{r+1}$ cannot certify a Normal Proposal. Thus, if $TC_r$ exists but $L_{r+1}$ does not enter $r+2$ via it, then $QC_{r+1}$ cannot certify either a Normal Proposal or a Fallback Recovery Proposal. Therefore $QC_{r+1}$ cannot exist. Consequently, no honest process will be able to enter $r' > r+2$ before $t_2+\tau$. However, by Lemma 7, all honest processes are guaranteed to enter $r+2$ or higher before $t_2+2\Delta$. Therefore, since $\tau > 4\Delta$ and because we have assumed that no honest process enters $r+2$ via $QC_r$, all honest processes must enter $r+2$ via $TC_r$, contradicting the former conclusion that $L_{r+1}$ must not do so. ■

**Lemma 14** (UL is LDC)**.** *If $QC_r$ is UL, $\tau > 4\Delta$ and $L_{r+1}$ is honest, then every honest process will LDC a block $B_{r'}$ with $B_r \Longleftarrow^* B_{r'}$ where $r' \geq r$ before $t_2+3\Delta$.*

*Proof:* By Definition 9, $QC_r$ must be locked by all honest processes. Therefore, by the Lock Rule, all honest processes must observe $QC_r$ whilst in $r+1$ or lower and, by Lemma 1, before sending $T_r$. Thus, $TC_r$ cannot exist. Consequently, since by Corollary 2 all honest processes will continue to enter new rounds after GST, the first honest process to enter $r+2$ must do so via $QC_r$ at $t_2 < t_1+\tau$. Therefore, by the QC Sync Rule, all honest processes will do the same before $t_2+\Delta$. Moreover, by Lemmas 9 and 12, $L_{r+1}$ will propose and all honest processes will observe this proposal both before they broadcast $T_{r+1}$ and before $t_2+2\Delta$. More precisely, since $TC_r$ cannot exist, by the QC Extension Rule, $L_{r+1}$ will create a Normal Proposal extending the block certified by $QC_r$ no later than the time that it observes this QC and thus before $t_2+\Delta$ Thus, since all honest processes will enter $r+2$ when they lock $QC_r$, by the Normal Vote Rule, if they remain in $r+2$ when they receive $L_{r+1}$'s proposal then they will send a Prepare vote for it. Consequently, all honest processes will observe $QC_{r+1}$ before $t_2+3\Delta$ and hence, having already observed $QC_r$, by the Two-Chain Commit Rule and Definition 6, will LDC $B_r$ if they have not already LDC a block for a higher round. ■

**Lemma 15** (UL or LDC)**.** *Let $L_r$ and $L_{r+1}$ be consecutive honest leaders and suppose that GST had passed before the first honest process entered $r$. If $\tau > 4\Delta$ then* *either $B_{r+1}$ will be UL before $t_2+4\Delta$ or every honest process will LDC a block for $r' \geq r-1$ before $t_2+5\Delta$.*

*Proof:* Suppose that $QC_{r+1}$ does not become UL and that at least one honest process does not LDC a block for $r' > r$ before the aforementioned intervals. However, Lemma 12 proves that $L_r$ will propose. Moreover, if this proposal becomes certified and is subsequently locked by all honest processes, then Lemma 14 shows that every honest process will LDC a block $B_{r'}$ with $B_r \Longleftarrow^* B_{r'}$ where $r' \geq r$ before $t_2+3\Delta$.

Suppose, then, that at least one honest process, say $v'$, fails to lock $QC_r$. However, if this happens because no honest process enters $r+2$ via $QC_r$ then, by Lemmas 11 and 13, either $QC_{r+1}$ will be UL before $t_2+4\Delta$ or every honest process will LDC a block $B_{r'}$ with $B_r \Longleftarrow^* B_{r'}$ where $r' \geq r+1$ before $t_2+5\Delta$. Therefore, at least one honest process must enter $r+2$ via $QC_r$. However, Lemma 11 proves that if $QC_r$ were for a Fallback Recovery Proposal then either all honest processes would have locked $QC_r$ before $t_1+4\Delta$, or every honest process would LDC a block for $r' \geq r$ before $t_1+5\Delta$, so $QC_r$ cannot certify a Fallback Recovery Proposal.

Thus, $QC_r$ must certify a Normal Proposal and at least one honest process must use it to enter $r+2$. Therefore, by Lemma 2, $TC_{r-1}$ cannot exist. Furthermore, by the Normal Vote Rule, at least $f+1$ honest processes must have locked $QC_{r-1}$ before voting for $B_r$, which must certify the parent of $B_r$. Additionally, since both $QC_{r-1}$ and $QC_r$ exist and because $B_{r-1} \Longleftarrow B_r$, by the Two Chain Commit Rule and Definition 6, $B_{r-1}$ satisfies the requirements to be LDC. Therefore, by the Garbage Collection Rule and Two-Chain Commit Rule, if any honest process observes $QC_r$ before it LDCs a block for a round higher than $r-1$, then it will LDC $B_{r-1}$. Moreover, by the QC Sync Rule, all honest processes will do the same if they have also not LDC a block for a higher round before they receive $QC_r$. However, we have already concluded that at least one honest process, say $v''$, must enter $r+2$ via $QC_r$ and that at least $f+1$ honest processes must have locked $QC_{r-1}$. Moreover, by Lemma 7, $v''$ must enter $r+2$ before $t_2+2\Delta$, so by the QC Sync Rule, all honest processes will observe this QC before $t_2+3\Delta$. Additionally, since this QC cannot exist before the aforementioned $f+1$ honest processes lock $QC_{r-1}$, all honest processes will also observe $QC_{r-1}$ before this time. Thus, all honest processes will LDC $B_{r-1}$ or higher before $t_2+3\Delta$. Moreover, by Corollary 1, every LDC block for $r' > r-1$ will have $B_{r-1} \Longleftarrow^* B_{r'}$.

Therefore, either $QC_{r+1}$ will be UL before $t_2+4\Delta$ or every honest process will LDC a block for $r' \geq r-1$ before $t_2+5\Delta$. ■

Corollary 3 follows from Lemmas 14 and 15.

**Corollary 3** (All LDC). *Let $L_r$ and $L_{r+1}$ be consecutive honest leaders and suppose GST had passed before the first honest process entered $r$. If $\tau > 4\Delta$ then all honest processes either LDC a block for $r' \geq r-1$ before $t_2+5\Delta$ or LDC a block for $r' \geq r+1$ before $t_3+3\Delta$.*

Corollary 3 is sufficient to allow us to complete the proof for Theorem 2. However, Chained Moonshot is guaranteed to LDC honest blocks under other circumstances as well. Since we are interested in properly understanding the round liveness properties of Chained Moonshot, we go on to present several more lemmas before completing our proof. This also allows us to make our bound on $c$, which indicates the minimum duration of network synchrony that a blockchain-based SMR protocol can tolerate whilst still achieving the Liveness property, tight.

Corollary 4 follows from Lemmas 14 and 15, and from Definition 4, which implies that there are guaranteed to be at least $kf$ pairs of consecutive honest leaders every $kn$ rounds.

**Corollary 4** (LDC Bounds). *If $\tau > 4\Delta$ and $L$ is deterministically fair then Chained Moonshot produces at least $kf$ honest blocks that satisfy the requirements to be LDC every $kn$ rounds after GST.*

**Lemma 16** (Canonical Progress). *Let $L_{r-1}$ and $L_r$ be consecutive leaders and let $L_r$ be honest. Also, suppose that GST had passed before the first honest process entered $r$. If $\tau > 4\Delta$ then the first certified block $B_{r'}$ with $r' > r$ extends either $B_{r-1}$ or $B_r$.*

*Proof:* Suppose that $B_{r'}$ does not extend either $B_{r-1}$ or $B_r$. However, if $B_{r'}$ were proposed as a Normal Proposal then by virtue of its definition as being the first certified block after round $r$ and by the SupraBFT Normal Vote Rule, $r' = r+1$ and $B_r \Longleftarrow B'$. Therefore, $B_{r'}$ must be proposed as a Fallback Recovery Proposal.

Lemma 12 proves that $L_r$ will propose. Consider the type of this proposal.

Suppose $L_r$ sends a Fallback Recovery Proposal containing $B_r$. Therefore, by Lemma 11, either $QC_r$ will be UL before $t_1+4\Delta$ or every honest process will LDC a block for $r' \geq r$ before $t_1+5\Delta$. In the latter case, by Lemma 6, the proof is complete. In the former case, by Definition 9 and Lemma 1, no honest process will ever send $T_r$, so $TC_r$ will never exist. Therefore, because

$B_{r'}$ must be proposed as a Fallback Recovery Proposal and thus must be justified by a $TC_{r'-1}$, $r' > r+1$. Furthermore, since $B_{r'}$ is the first certified block for a round greater than $r$, $TC_{r'-1}$ cannot contain a greater QC than $QC_r$. Moreover, because all honest processes will lock $QC_r$, which they can only do whilst in $r+1$, and because $r'-1 \geq r+1$, by Lemma 5, every honest $T_{r'-1}$ message is guaranteed to contain $QC_r$. Consequently, $TC_{r'-1}$ will necessarily include the QC for $B_r$ as its highest QC. Thus $B_r \Longleftarrow B_{r'}$.

Alternatively, suppose that $L_r$ does not make a Fallback Recovery Proposal. Therefore, by the Fallback Proposal Rule, $L_r$ must not enter $r+1$ via $TC_{r-1}$. Suppose, then, that $L_r$ enters $r+1$ via $QC_{r-1}$. Therefore, by Lemma 7, it must do so before $t_1+2\Delta$. Consequently, all honest processes are guaranteed to observe $QC_{r-1}$ before $t_1+3\Delta$ and thus will add it to $U$ upon doing so. However the Timeout and Timeout Sync Rules together ensure that no honest process will broadcast $T_r$ before $t_1+\tau$. Therefore, by Lemma 4, neither can any honest process have sent a Timeout message for a higher round before this time. Hence, since $\tau > 4\Delta$, by Lemma 5, if any honest process sends a Timeout message for $r$ or higher then this message will contain $QC_{r-1}$ or higher. However, as previously observed, $TC_{r'-1}$ cannot contain greater QC than $QC_r$. Thus, either $B_{r-1} \Longleftarrow B_{r'}$ or $B_r \Longleftarrow B_{r'}$.

Otherwise, by Corollary 2, $L_r$ must proceed directly from $r$ to $r_h > r+1$. However, by Lemma 4, at least one honest process must still enter $r+1$. Consequently, by Lemma 7, $L_r$ must enter $r_h$ before $t_1+2\Delta$. Moreover, since $\tau > 4\Delta$, no honest process can have sent a Timeout message for $r$ before this time, nor, by Lemma 4, for any higher round. Thus, $L_r$ must enter $r_h$ via $QC_{r_h-2} \geq QC_r$. Therefore, by the QC Sync Rule, all honest processes will observe $QC_{r_h-2}$ before $t_1+3\Delta$, so, by Lemma 5, every honest Timeout message for $r$ or higher will contain at least $QC_{r_h-2}$. However, we have already concluded that $TC_{r'-1}$ cannot contain a greater QC than $QC_r$. Thus, $QC_{r_h-2} = QC_r$. Therefore, $B_r \Longleftarrow B_{r'}$.

Thus, in all cases, either $B_{r-1} \Longleftarrow B_{r'}$ or $B_r \Longleftarrow B_{r'}$. ■

**Lemma 17** (LC Bounds). *If $\tau > 4\Delta$ and $L$ is deterministically fair then all honest processes will LC at least $k(2f+1)$ new blocks, $k(f+1)$ of which will be honest, every $kn$ rounds after GST.*

*Proof:* Lemma 16 proves that for each pair of Byzantine and honest leaders, every subsequently certified block is a descendent of a block proposed by one of these two processes. Therefore, by Definition 5, the successful block is *canonical*. Additionally, by Lemma 15, whenever $L$ elects two consecutive honest leaders, say $L_r$ and $L_{r+1}$, either $B_{r+1}$ will be UL or every honest process will LDC a block for $r' \geq r-1$ before $t_2+5\Delta$. In the former case, by Lemma 14 and Definition 5, $B_{r+1}$ will be canonical. In the latter case, by Lemma 6, the LDC block will be canonical. Moreover, since we have assumed that $L$ is deterministically fair, Definition 4 implies that the adversary controls the leaders of $kf$ rounds every $kn$ rounds. Consequently, the adversary can prevent at most $kf$ blocks from becoming canonical in the same interval. Thus, Chained Moonshot is guaranteed to produce at least $k(2f+1)$ canonical blocks every $kn$ rounds, at least $k(f+1)$ of which will be honest. Additionally, by Corollary 3, every time $L_r$ and $L_{r+1}$ are both honest, all honest processes will LDC a new block before $t_3+3\Delta$. Therefore, all honest processes will LC at least $k(2f+1)$ new blocks, $k(f+1)$ of which will be honest, every $kn$ rounds after GST. ■

Let $\tau = x\Delta$ where $x > 4$ and suppose $M = c\Delta$. Recall also that Lemma 8 proves that all honest processes observe a QC for given round no later than $4\Delta+\tau$ after the first honest process enters it. Consequently, the upper bound on the length of any round is $(x+4)\Delta$. Therefore, if $u = x+4$ then if $c > uj+l \mid j > 1, l > 0$ then Theorem 2 follows for all variants of Chained Moonshot that have:

1) Leader election functions that deterministically elect at least one pair of consecutive honest leaders every $j$ rounds.
2) Block delivery protocols that guarantee that if any honest process observes a QC for some block $B$ at time $t$ then every honest process will receive $B$ before $t+l\Delta$.

**Theorem 2** (Liveness). *For every run $R \in \mathcal{R}_{\mathcal{P}}$, for each synchronous interval $S \in C_R(M)$, each honest process $v \in \mathcal{V}$ appends at least $\lfloor \frac{|S|}{M} \rfloor$ new blocks proposed by honest leaders to its local blockchain $\mathbf{B_v}$ during $S$.*

*Proof:* Recall that $C_R(M) = \{S \mid S \in S_R$ and $|S| \geq M\}$ where $M = c\Delta$. Therefore, since every $S$ occurs after GST, and because $|S| \geq (uj+l)\Delta$ where $j > 1$ and $k > 0$, Corollary 2 proves that all honest processes continue to enter increasing rounds during each $S$.

Furthermore, since $|S| \geq (uj+l)\Delta$, every honest process is guaranteed to advance through at least $j$ rounds during each period of synchrony during $R$. Consequently, because $L$ guaranteed to elect at least one pair of consecutive honest leaders every $j$ rounds, Corollary 3 implies that all honest processes will LC at least one new block whenever this occurs.

Finally, since all in-transit messages are delivered upon GST and because every $S$ occurs after GST, the lossless network and the QC Sync Rule together ensure that every honest process will have observed the QCs of all previously-certified blocks before scheduling $B_r$ for commit. Therefore, if QCs include the blocks themselves per the simplified algorithm presented in Section IV, then by the semantics of the $commit$ function given in Table II, every honest process will append $B_r$ and its uncommitted ancestors to its local blockchain no later than the time that they observe the QC for $B_{r+1}$ (i.e. when they LC $B_r$).

Alternatively, if QCs instead include the digest of the corresponding block then some honest processes may need to retrieve some blocks via the synchronisation protocol discussed in Section VI in order to append it to $\mathbf{B_v}$. However, recall that we are assuming that the protocol implementation incorporates a block delivery protocol that ensures that if any honest process observes a QC for some block $B$ at time $t$ then every honest process will receive $B$ before $t + l\Delta$. Therefore, since $|S| \geq (uj + l)\Delta$ and because all honest processes must observe the QC for $B_r$ before $u(j-1)\Delta$ in order to lock it, which they must have done in order to vote for $B_{r+1}$, they are all guaranteed to receive $B_r$ and by extension, all of its ancestors, during $S$ and thus will append $B_r$ and its uncommitted ancestors to their respective local blockchains during $S$.

By extension, if $|S| = qM + r$ where $q > 0$, $r \geq 0$, then each honest process $v \in \mathcal{V}$ will append at least $q$ new blocks proposed by honest leaders to its local blockchain $\mathbf{B_v}$ during $S$. Thus, each honest process $v \in \mathcal{V}$ appends at least $\lfloor \frac{|S|}{M} \rfloor$ new blocks proposed by honest leaders to its local blockchain $\mathbf{B_v}$ during $S$. ■

We observe that Chained Moonshot with a round-robin leader election function satisfies Theorem 2 for $j = f+2$ when paired with a block delivery protocol that provides the aforementioned guarantee. Such a leader election function is, per Definition 4, deterministically fair with $k = 1$ and thus ensures that at least $2f + 1$ out of every $n$ leaders is honest. This ensures that the protocol will have at least one sequence of two consecutive honest leaders every $f + 2$ rounds. Moreover, if the accompanying block delivery protocol is the simple synchronisation protocol discussed in Section VI, then the implementation of Chained Moonshot satisfies Theorem 2 for $l = 2$.

TABLE V
BENCHMARK NETWORK DISTRIBUTION

| Network Size | Regions |
|---|---|
| 10, 50 | us-east-1, us-west-1, eu-north-1, ap-northeast-1, ap-southeast-2 |
| 100, 200 | us-east-1, us-east-2, us-west-1, us-west-2, ap-east-1, ap-south-1, ap-northeast-1, ap-northeast-2, ap-northeast-3, ap-southeast-1, ap-southeast-2, ap-southeast-3, ca-central-1, eu-central-1, eu-west-1, eu-west-2, eu-west-3, eu-north-1, eu-south-1, me-south-1 |

Finally, we observe that the previously given value of $M$ is over-approximate. It assumes that the network takes at least $u\Delta$ to decide on a value every round, but we know from the former reasoning that at most $kf$ out of every $kn$ rounds can end with a TC when $L$ is deterministically fair. Consequently, a tighter bound on the required length of each $S \in C_R(M)$ exists, although we do not derive it here.

## VIII. EVALUATION

We presented a brief and informal theoretical comparison between Chained Moonshot and some of its recent blockchain-based SMR predecessors in Section III. We reserve a more detailed comparison for a later version of this paper and now go on to discuss our practical evaluation of our protocol.

As mentioned in Section III, Jolteon was the most efficient derivative of Chained HotStuff known to us during our development of Chained Moonshot. Jolteon also has multiple publicly-available implementations, making it a convenient candidate for comparison. Accordingly, we implemented Chained Moonshot including the optimisation discussed in Section VI by modifying the code for Jolteon available in the Narwhal-HotStuff branch of the repository [1] created by Facebook Research for evaluating Narhwal and Tusk. We decoupled our implementation from Narhwal and then did the same for Jolteon 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 block proposers of each protocol create parametrically sized payloads during the block creation process. 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.

Our goal was to compare the throughput and latency of Chained Moonshot and Jolteon across two dimensions:

firstly, with respect to the size of the network; and secondly, with respect to the size of the block payload. We established two metrics for throughput: Firstly, the number of blocks committed by at least $2f+1$ processes in the network during a run, hereafter referred to as *block throughput*; and secondly, the average number of bytes of payload data transferred per second during the run, hereafter referred to as *transfer rate*. We chose these metrics for throughput rather than the typical transactions committed per second, because our protocol is agnostic to the transaction delivery and execution layers. Correspondingly, for latency, we measured the time between the creation of a block and its commit by the $2f+1th$ process. We likewise chose this metric rather than the typical end-to-end metric, which instead measures the time between the client's submission of a transaction and its receipt of proof of the transaction's successful execution, for the same reason.

In accordance with the brief analysis of Jolteon's latency given in Section III (i.e. $5\delta$ vs Chained Moonshot's $3\delta$), our hypothesis was that our implementation would exhibit $40\%$ lower latency than Jolteon when Proposal and Prepare dissemination times were approximately equal, with this improvement decreasing towards $30\%$ lower latency as the payload size increased and thus the Proposal transmission time increased relative to the Prepare transmission time. We likewise expected it to produce twice the throughput when the dissemination times of these two types of messages were equal, decreasing towards equal throughput as the Proposal transmission time relatively increased. Both of these expectations were subject to the assumption that the increased communication cost incurred by Chained Moonshot's broadcasting of QCs and Prepare votes would remain within the network and computational bandwidth of the nodes.

We tested network sizes of 10, 50, 100 and 200 nodes, and seven payload sizes ranging between 1.8kB and 18MB, where individual payload items were 180 bytes in size. We incremented the payload size by an order of magnitude after each test in order to quickly identify the approximate transfer rate limit of each protocol in the larger networks. We chose the upper bound of 18MB (except for the Throughput vs Latency experiments) to avoid the excessively high latencies exhibited by the larger networks obscuring the visualisation of the other results. Likewise, we split the final interval between 10k payload items and 100k payload items into a further two intervals to increase the precision of the results reported in Figure 5. We executed each combination of network and payload size three times to increase the representativity of the results, with runs lasting five minutes each. The reported result for each of these configurations was calculated as the average of the three runs.

Our networks were constructed from m5.xlarge AWS EC2 instances running Ubuntu 20.04. Each of these instances had a network bandwidth of up to 10Gbps[3], 16GB of memory and Intel Xeon Platinum 8000 series processors with 4 virtual cores. The instances were distributed across the regions described in Table V. We configured all nodes to be honest because we were primarily interested in showing Chained Moonshot's benefits under typical operating conditions. We intend to report on its behaviour under adversarial conditions in future work. The timeout interval $\tau$ was set to five seconds for all configurations up to the 1.8MB payload size. Thereafter, we had to adjust this value upward to enable the protocols to continue to make progress. Since we were focused on testing the Normal Paths of these protocols, this value is not particularly relevant to the results, but we report it for the sake of completeness.

As shown by Figures 3 and 4, Chained Moonshot outperformed Jolteon in both latency and throughput in every configuration, averaging $41.1\%$ lower latency and $54.9\%$ higher throughput across all configurations.

Most configurations exhibited relatively consistent latency improvements, but the general trend did not conform to our expectations. Specifically, Chained Moonshot consistently produced its smallest improvements for the 1.8kB payload size, which should have had similar dissemination times for both Proposals and Prepares and thus, according to our simple analysis, should have produced the greatest improvement. Comparatively, increasing the payload size did little to reduce Chained Moonshot' outperformance, with every other payload size averaging at least $39\%$ lower latency than Jolteon across all network sizes, with many configurations exceeding the maximum expected improvement of $40\%$, implying that its vote-broadcasting incurs negligible overhead under these configurations. The minimum of $22.8\%$ decreased latency relative to Jolteon was produced by the 50 node, 1.8kB configuration, while the maximum decrease of $58.7\%$ was seen in the 200 node, 180kB configuration.

Chained Moonshot's relative improvement in throughput compared to Jolteon generally increased with the size of the network, increasing from $38.2\%$ better on average for the 10 node network to $65.7\%$ better on average

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

Fig. 3. Latency

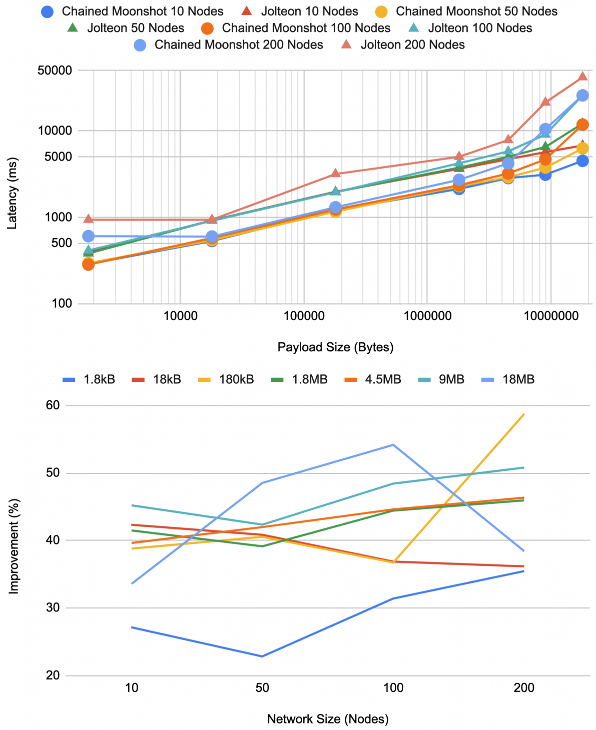
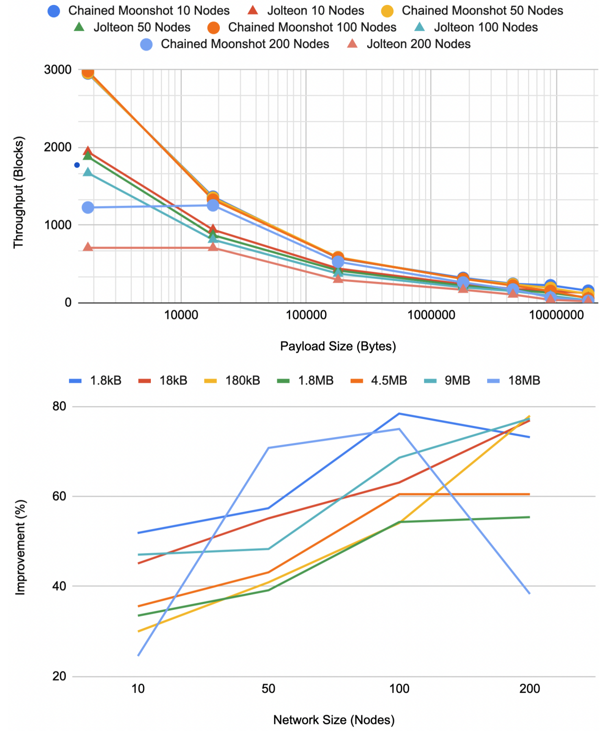



Fig. 4. Throughput




in the 200 node network. However, its increase in throughput was generally much less than expected for the smaller payload sizes and fluctuated as the payload size increased rather than decreasing as expected. Chained Moonshot produced a maximum throughput increase of 78.4% for the 100 node, 1.8kB configuration, with the minimum of 24.5% coming from the 10 node, 18MB configuration.

Latency roughly doubled and block throughput approximately halved for both protocols for every order-of-magnitude increase in payload size. By contrast, both metrics remained comparable as the network size increased for the smaller payload sizes, with the payload sizes above 1.8MB consistently producing worse performance for both metrics as the network size increased.

Figure 5 presents the characteristic curve for the transfer rate and latency of the two protocols in two graphs, the first of which uses standard scaling, and the second of which uses log scaling on both axes to emphasise the separation between Chained Moonshot and Jolteon at smaller payload sizes. As expected, both protocols showed both an increasing transfer rate and increasing commit latency as the payload size increased, but eventually reached a point of saturation after which

TABLE VI

THROUGHPUT VS LATENCY: INFLECTION POINTS

| Network | Payload Size | Transfer Rate | Latency |
|---|---|---|---|
| **Chained Moonshot** | | | |
| 10 | 4.5MB | 6.8MB/s | 3.1s |
| 50 | 9MB | 5.8MB/s | 3.8s |
| 100 | 4.5MB | 3.4MB/s | 3.2s |
| 200 | 4.5MB | 2.6MB/s | 4.2s |
| **Jolteon** | | | |
| 10 | 18MB | 7.6MB/s | 6.8s |
| 50 | 9MB | 3.9MB/s | 6.5s |
| 100 | 4.5MB | 2.3MB/s | 5.8s |
| 200 | 4.5MB | 1.6MB/s | 7.9s |

increasing the payload size decreased the transfer rate but continued to increase the latency. Both protocols reached saturation in the 100 and 200 nodes networks, with both recording their maximum transfer rates under the 9MB payload in the 100 node network and the 4.5MB payload in the 200 node network. Jolteon maximised its transfer rate at 2.7MB/s in the 100 node network, compared to the 4.6MB/s of Chained Moonshot. Comparatively, Chained Moonshot produced 2.6MB/s in the 200 node network, while Jolteon achieved only 1.6MB/s. Neither protocol reached saturation in the 50 and 10 node

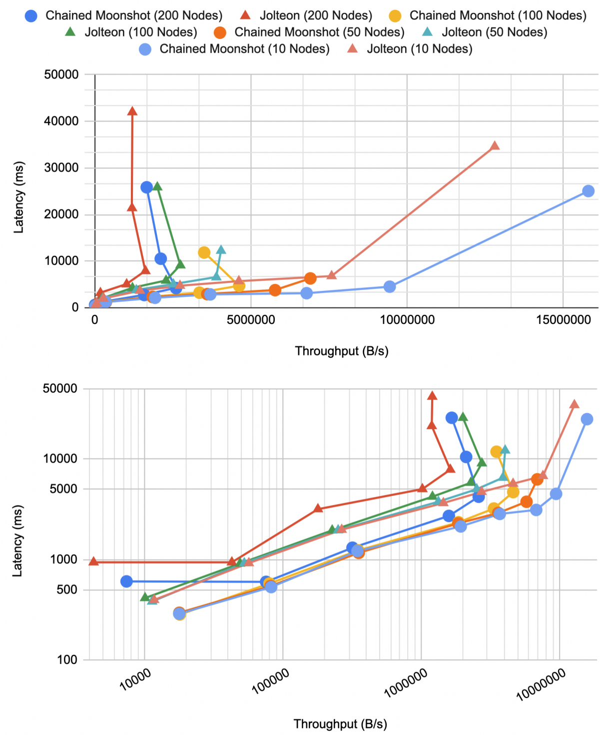



Fig. 5. Throughput vs Latency

networks under the tested payload sizes. We added a 180MB payload test to the 10 node network in order to clarify the trend in this setting. As the figures show, Chained Moonshot consistently outperformed Jolteon in all configurations.

Table VI summarises the inflection points of the various curves. These points identify the payload size under which the respective protocol maximised its transfer rate compared to its commit latency and thus represent the point of maximum efficiency, out of the tested configurations, for the given network size. Overall, Chained Moonshot achieved both higher throughput and lower latency when performing optimally in all network sizes except for the 10 node network, where it produced a slightly lower transfer rate than Jolteon, but was able to do so with less than half the latency.

In summary, we attribute the aforementioned deviations from our expectations to the simplicity of our analytical model, which did not precisely factor in either the many nuances of the two implementations or the system's actual bandwidth and compute resources. Furthermore, Chained Moonshot exhibited a higher variance between runs than Jolteon did, for both metrics and across most configurations, possibly indicating that there remains room for improvement in our implementation. Overall, these results show that Chained Moonshot provides meaningfully decreased latency and increased throughput compared to Jolteon across all tested configurations.

## IX. Conclusion

We introduced Moonshot, a new family of blockchain-based BFT SMR protocols characterised by optimistic proposal. We also formally defined Chained Moonshot, a variant of Moonshot that leverages QC chaining and vote-broadcasting to achieve a best-case block finalisation latency of $3\delta$ and block period of $\delta$ at the cost of a best-case communication complexity of $O(n^2)$ messages per decision. This corresponds to an expected $40\%$ reduction in block finalisation latency and a $50\%$ reduction in block period with respect to Jolteon's normal path.

In our experiments, Chained Moonshot exhibited an average of $41.1\%$ lower block finalisation latency and $54.9\%$ higher block throughput when compared to Jolteon in WANs of 10, 50, 100 and 200 nodes with varying payload sizes. We intend to perform further experiments with Chained Moonshot to showcase its performance in the fallback path. We will also be updating this paper in the near future to include formal descriptions and analyses of other variants of Moonshot.

## References

[1] Narwhal-HotStuff Github Repository. https://github.com/facebookresearch/narwhal/tree/narwhal-hs. [Online; accessed 22-January-2023].

[2] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on bft consensus, 2018.

[3] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *3rd Symposium on Operating Systems Design and Implementation (OSDI 99)*, New Orleans, LA, February 1999. USENIX Association.

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

[5] Luciano Freitas, Andrei Tonkikh, Adda-Akram Bendoukha, Sara Tucci-Piergiovanni, Renaud Sirdey, Oana Stan, and Petr Kuznetsov. Homomorphic sortition – secret leader election for pos blockchains, 2022.

[6] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback, 2021.

[7] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, jul 1982.

[8] Fred B. Schneider. The state machine approach: A tutorial. In Barbara Simons and Alfred Spector, editors, *Fault-Tolerant Distributed Computing*, pages 18–41, New York, NY, 1990. Springer New York.

[9] The Diem Team. Diembft v4: State machine replication in the diem blockchain, 2021.

[10] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 347–356, New York, NY, USA, 2019. Association for Computing Machinery.