State Machine Replication in the Libra Blockchain

Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, Alberto Sonnino*

Abstract. This report presents LibraBFT, a robust and efficient state machine replication system designed for the Libra Blockchain. LibraBFT is based on HotStuff, a recent protocol that leverages several decades of scientific advances in Byzantine fault tolerance (BFT) and achieves the strong scalability and security properties required by internet settings. LibraBFT further refines the HotStuff protocol to introduce explicit liveness mechanisms and provides a concrete latency analysis. To drive the integration with the Libra Blockchain, this document provides specifications extracted from a fully-functional simulator. These specifications include state replication interfaces and a communication framework for data transfer and state synchronization among participants. Finally, this report provides a formal safety proof that induces criteria to detect misbehavior of BFT nodes, coupled with a simple reward and punishment mechanism.

1. Introduction

The advent of the internet and mobile broadband has connected billions of people globally, providing access to knowledge, free communications, and a wide range of lower-cost, more convenient services. This connectivity has also enabled more people to access the financial ecosystem. Yet, despite this progress, access to financial services is still limited for those who need it most.

Blockchains and cryptocurrencies have shown that the latest advances in computer science, cryptography, and economics have the potential to create innovation in financial infrastructure, but existing systems have not yet reached mainstream adoption. As the next step toward this goal, we have designed the Libra Blockchain [1], [2] with the mission to enable a simple global currency and financial infrastructure that empowers billions of people.

At the heart of this new blockchain is a consensus protocol called LibraBFT — the focus of this report — by which blockchain transactions are ordered and finalized. LibraBFT decentralizes trust among a set of validators that participate in the consensus protocol. LibraBFT guarantees consensus on the history of transactions among honest validators and remains safe even if a threshold of participants are Byzantine (i.e., faulty or corrupt [3]). By embracing the classical approach to Byzantine fault tolerance, LibraBFT builds on solid and rigorously proven foundations in distributed computing. Furthermore, the scientific community has made steady progress, which LibraBFT builds on, in scaling consensus technology and making it robust for internet settings.

Initially, the participating validators will be permitted into the consensus network by an association consisting of a geographically distributed and diverse set of Founding Members, which are organizations chosen according to objective membership criteria with a vested interest in bootstrapping the Libra ecosystem [2]. Over time, membership eligibility will shift to become open and based only on an organization's holdings of Libra [4].

The LibraBFT consensus is based on a cutting-edge technique called HotStuff [5], [6] that bridges between the world of BFT consensus and blockchain. This choice reflects vast expert knowledge and exploration of various alternatives and provides LibraBFT with the following key properties that are crucial for decentralizing trust:

- **Safety:** LibraBFT maintains consistency among honest validators, even if up to one-third of the validators are corrupt.
- **Asynchrony:** Consistency is guaranteed even in cases of *network asynchrony* (i.e., during periods of unbounded communication delays or network disruptions). This reflects our belief that building internet-scale consensus protocol whose safety relies on synchrony would be inherently both complex and vulnerable to Denial-of-Service (DoS) attacks on the network.
- **Finality:** LibraBFT supports a notion of *finality*, whereby a transaction becomes irreversibly committed. It provides concise commitments that authenticate the result of ledger queries to an end user.
- **Linearity and Responsiveness:** LibraBFT has two desirable properties that BFT consensus protocols preceding HotStuff were not able to simultaneously support — *linearity* and *responsiveness*. These two technical concepts are linked with the notion of *leaders*, a key approach for driving progress against partial synchrony. Informally, linearity guarantees that driving transaction commits incurs only linear communication (this is optimal) even when leaders rotate; responsiveness means that the leader has no built-in delay steps and advances as soon as it collects responses from validators.
- **Simplicity and Modularity:** The core logic of LibraBFT allows simple and robust implementation, paralleling that of public blockchains based on Nakamoto consensus [7]. Notably, the protocol is organized around a single communication phase and allows a concise safety argument.
- **Sustainability:** Current public blockchains, where trust is based on computational power, have been reported to consume vast amounts of energy [8] and may be subject to centralization [9]. LibraBFT is designed as a proof-of-stake system, where participation privileges are granted to known members based on their financial involvement. LibraBFT can support economic incentives to reward good behaviors and/or punish wrongdoings from stakeholders. Computational costs in LibraBFT consist primarily of cryptographic signatures, a standard concept with efficient implementations.

Key technical approach. LibraBFT is a consensus protocol that progresses in rounds, where in each round a leader is chosen amongst the validators. As mentioned above, this key approach is needed for driving progress against partial synchrony. The leader proposes a new block consisting of transactions and sends it to the rest of the validators, who approve the new block if it consists of valid transactions. Once the leader collects a majority of votes, she sends it to the rest of the validators. If a leader fails to propose a valid block or does not aggregate enough votes, a timeout mechanism will force a new round, and a new leader will be chosen from the validators. This way, new blocks extend the blockchain. Eventually, a block will meet the *commit rule* of LibraBFT, and once this happens, this block and any prior block is committed.

Related work. A comprehensive survey is beyond the scope of this manuscript (see for example [10]–[12]). Here we mention key concepts and mechanisms that influenced our work.

Consensus algorithms in classical setting. The Byzantine consensus problem was pioneered by Lamport et al. [3], who also coined the term Byzantine to model arbitrary, possibly maliciously corrupt behavior. The safety of the solution introduced by Lamport et al. relied on synchrony, a dependency that practical systems wish to avoid both due to complexity and because it exposes the system to DoS attacks on safety.

In lieu of synchrony assumptions, randomized algorithms, pioneered by Ben-Or [13], guarantee progress with high probability. A line of research gradually improved the scalability of such algorithms, including [14]–[17]. However, most practical systems did not yet incorporate randomization. In the future, LibraBFT may incorporate certain randomization to thwart adaptive attacks.

A different approach for asynchronous settings, introduced by Dwork et al. [18], separated safety (at all times) from liveness (during periods of synchrony). Dwork et al. introduced a round-by-round paradigm where each round is driven by a designated leader. Progress is guaranteed during periods of synchrony as soon as an honest leader emerges, and until then, rounds are retired by timeouts. Dwork et al.'s approach (DLS) underlies most practical BFT works to date, with steady improvements to its performance. Specifically, it underlies the first practical solution introduced by Castro and Liskov [19] called PBFT. In PBFT, an honest leader reaches a decision in two all-to-all communication rounds. In addition to the original open-source implementation of PBFT, the protocol has been integrated into BFT-SMaRt [20] and, recently, into FaB [21]. Zyzzyva [22] adds an optimistically fast track to PBFT that can reach a decision in one round when there are no failures. An open-source implementation of Zyzzyva was built in Upright [23]. Response aggregation using threshold cryptography was utilized in consensus protocols by Cachin [24] and Reiter [25] to replace all-to-all communication with an all-to-collector and collector-to-all pattern that incurs only linear communication costs. Threshold signature aggregation has been incorporated into several PBFT-based systems, including Byzcoin [26] and SBFT [27]. Similarly, LibraBFT incorporates message collection and fast signature aggregation [28]. Compared with threshold signature, signature aggregation in LibraBFT does not require distributed setup and enables economic incentives for voters at the price of one additional bit per node per signature.

Two blockchain systems, Tendermint [29] and Casper [30], presented a new variant of PBFT that simplifies the leader-replacement protocol of PBFT such that it has only linear communication cost (*linearity*). These variants forego a hallmark property of practical solutions called *responsiveness*. Informally, (optimistic) responsiveness holds when leaders can propose new blocks as soon as they receive a fixed number of messages, as opposed to waiting for a fixed delay. Thus, Tendermint and Casper introduced into the field a trade-off in practical BFT solutions — either they have linearity or responsiveness, but not both. The HotStuff solution, which LibraBFT is based on (as well as other recent blockchains, notably ThunderCore with a variant named PaLa [31]) resolved this trade-off and presented the first BFT consensus protocol that has both.

Consensus in a permissionless setting. All the works mentioned above assume a permissioned setting, i.e., the participating players are known in advance. Differently, in a permissionless setting, any party can join and participate in the protocol — which is what Nakamoto Consensus (NC) [7] aims to solve — resulting in an entirely different protocol structure. In NC, transactions (gathered in blocks) are chained and simply disseminated to the network with a proof of work. Finality is defined probabilistically — the probability that a block remains in the history is proportional to the computational cost of succeeding blocks in the blockchain. The reward mechanism for extending the current chain suffices to incentivize miners to accept the current chain de facto and rapidly converge on a single, longest fork. Casper and HotStuff exhibit similar simplicity of protocol structure. They embed the protocol rounds into a (possibly branching) chain and deduce commit decisions by simple offline analysis of the chains.

Several blockchains are similarly based on graphs of blocks in the form of direct acyclic graphs (DAG) allowing greater concurrency in posting blocks into the graph, e.g., GHOST [32], Conflux [33], Blockmania [34] and Hashgraph [35]. Our experience with some of these paradigms indicates that recovering graph information and verifying it after a participant loses connection temporarily can be challenging. In LibraBFT, only leaders can extend chains; hence, disseminating, recovering, and verifying graph information is simple and essentially linear.

Revisiting LibraBFT. LibraBFT leverages HotStuff (ArXiv version [5], to appear in PODC'19 [6]) and possesses many of the benefits achieved in four decades of works presented above. Specifically, LibraBFT adopts the DLS and PBFT round-based approach, has signature aggregation, and has both linearity and responsiveness. We also found that the chain structure of Casper and HotStuff leads to robust implementation and concise safety arguments.

Compared with HotStuff itself, LibraBFT makes a number of enhancements. LibraBFT provides a detailed specification and implementation of the pacemaker mechanism by which participants synchronize rounds. This is coupled with a liveness analysis that consists of concrete bounds to transaction commitment. LibraBFT includes a reconfiguration mechanism of the validator voting rights (epochs). It also describes mechanisms to reward proposers and voters. The specification allows deriving safe and complete criteria to detect validators that attempt to break safety, enabling punishment to be incorporated into the protocol in the future. We also elaborate on the protocol for data dissemination among validators to synchronize their state.

Organization. The remainder of this report is structured as follows: we start by introducing important concepts and definitions (Section 2) and how LibraBFT is used in the Libra Blockchain (Section 3). We then describe the core data types of LibraBFT and its network communication layer (Section 4). Next, we present the protocol itself (Section 5) with enough detail to prepare the proof of safety (Section 6). We then describe the pacemaker module (Section 7) and prove liveness (Section 8). Finally, we discuss the economic incentives of LibraBFT (Section 9) and conclude in Section 10.

In this initial report, we have chosen to use a minimal subset of Rust as a specification language for the protocol, whenever code was needed. We provide code fragments directly extracted from our reference implementation in a discrete-event simulated environment. We intend to share the code for this simulator and provide experimental results in a subsequent version of the report.

2. Overview and Definitions

We start by describing the desired properties of LibraBFT and how our state machine replication protocol is meant to be integrated into the Libra Blockchain.

2.1. State Machine Replication

State Machine Replication (SMR) protocols [36] are meant to provide an abstract state machine distributed over the network and replicated between many processes, also called *nodes*.

Specifically, a SMR protocol is started with some initial *execution state*. Every process can submit *commands* and observe a sequence of *commits*. Each commit contains the execution state that is the result of executing a particular command on top of the previous commit. Commands may be rejected during execution: in this case, they are called *invalid* commands.

Assuming that command execution is deterministic, we wish to guarantee the following properties:

(safety) All honest nodes observe the same sequence of commits.

(liveness) New commits are produced as long as valid commands are submitted.

Note that nodes should observe commits in the same order but not necessarily at the same time. The notion of an honest node is made precise below in Section 2.3.

2.2. Epochs

For practical applications, the set of nodes participating in the protocol can evolve over time. In LibraBFT, this is addressed by supporting a notion of *epoch*:

- Each epoch begins using the last execution state of the previous epoch — or using system-wide initial parameters for the first epoch.
- We assume that every execution state contains a value `epoch_id` that identifies the current epoch.
- When a command that increments `epoch_id` is committed, the current epoch stops after this commit, and the next epoch is started.

2.3. Byzantine Fault Tolerance

Historically, fault-tolerant protocols were meant to address common failures, such as crashes. In the context of a blockchain, the SMR consensus protocol is used to limit the power of individual nodes in the system. To do so, we must guarantee safety and liveness even when certain nodes deviate arbitrarily from the protocol.

In the rest of this report, we assume a fixed, unknown subset of malicious nodes for every epoch, called *Byzantine* nodes [3]. All the other nodes, called *honest nodes*, are assumed to follow protocol specifications scrupulously. During a given epoch, we assume that every SMR node α has a fixed *voting power*, denoted $V(\alpha) \geq 0$. We write N for the total voting power of all nodes and assume a *security threshold* f as a function of N such that $N > 3f$. For example, we may define $f = \lfloor \frac{N-1}{3} \rfloor$. For notational simplicity in this report, we refer to a voting power of x as consisting of x nodes.

We analyze all consensus properties in the context of the following *BFT assumption*:

> **(bft-assumption)** The combined voting power of Byzantine nodes during any epoch must not exceed the security threshold f.

A subset of nodes whose combined voting power M satisfies $M \geq N - f$ is called a *quorum*. The notion of quorum is justified by the following classic lemma [37]:

Lemma B1: Under BFT assumption, for every two quorums of nodes in the same epoch, there exists an honest node that belongs to both quorums.

We recall the proof of Lemma B1 in Section 6.1.

2.4. Cryptographic Assumptions

We assume a hash function and a digital signature scheme that are secure against computationally-bounded adversaries and require that every honest node keeps its private signature key(s) secret.

Since our protocol only hashes and signs public values, we may assume all digital signatures to be unforgeable in a strong non-probabilistic sense, meaning that any valid signature must originate from the owner of the private key. Similarly, we may assume that collisions in the hash function `hash` will never happen, therefore $\texttt{hash}(m_1) = \texttt{hash}(m_2)$ implies $m_1 = m_2$.

2.5. Networking Assumptions and Honest Crashes

While the safety of LibraBFT is guaranteed under the BFT assumption alone, liveness requires additional assumptions on the network and the processes. Specifically, we will assume that the network alternates between periods of bad and good connectivity, known as periods of *asynchrony* and *synchrony*, respectively. Liveness can only be guaranteed during long-enough periods of synchrony.

During periods of asynchrony, we allow messages to be lost or to take an arbitrarily long time. We also allow honest nodes to crash and restart. During periods of synchrony, we assume that there exists an upper bound δ_M to the transmission delay taken by any message between honest nodes; besides, honest processes must be responsive and cannot crash.

We must stress that the adversary controls malicious nodes and the scheduling of networking messages even during periods of synchrony, subject to the maximal delay δ_M.

The parameters of this model — such as the value of δ_M or whether the network is currently synchronous or not — are not available to the participants within the system. To simplify the analysis, it is usual in the literature to consider only two periods: before some unknown global stabilization time, called *GST*, and after GST. Our proof of liveness (Section 8) will give concrete upper bounds on the time needed by the system to produce a commit after GST.

More formally, the assumptions on network and crashes are written as follows:

> **(eventually-synchronous-network)** After GST, the network delivers all messages between honest nodes under some (unknown) time delay $\delta_M > 0$.
>
> **(eventually-no-crash)** After GST, honest nodes are perfectly responsive and never crash.

We remark that the GST model does not take into account CPU time associated with message processing. Besides, message sizes in LibraBFT are not bounded, and, therefore, a fixed δ_M is arguably over-simplified. In future work, we may enforce strict bounds on message sizes or see the networking delay for each message as the composition of a fixed latency and a delay of transmission proportional to the message size.

2.6. Leaders, Votes, Quorum Certificates

LibraBFT belongs to the family of *leader-based* consensus protocols. In leader-based protocols, validators make progress in rounds, each having a designated validator called a *leader*. Leaders are responsible for proposing new blocks and obtaining signed *votes* from the validators on their proposals. LibraBFT follows the chained variant of HotStuff [5], where a round is a communication phase with a single designated leader, and leader proposals are organized into a chain using cryptographic hashes. During a round, the leader proposes a block that extends the longest chain it knows. If the proposal is valid and timely, each honest node will sign it and send a *vote* back to the leader. After the leader has received enough votes to reach a quorum, it aggregates the votes into a *Quorum Certificate* (QC) that extends the same chain again. The QC is broadcast to every node. If the leader fails to assemble a QC, participants will timeout and move to the next round. Eventually, enough blocks and QCs will extend the chain in a timely manner, and a block will match the *commit rule* of the protocol. When this happens, the chain of uncommitted blocks up to the matching block become committed.

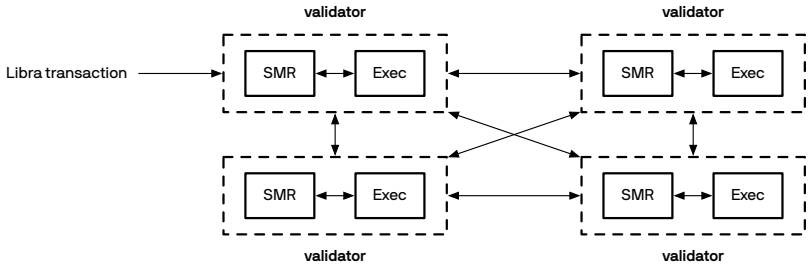

Figure 1: Integration of the SMR module into the Libra Blockchain

3. Integration with the Libra Blockchain

3.1. Consensus Protocol

We expect LibraBFT to be used in the Libra Blockchain [2] as follows:

- The *validators* of Libra participate in the LibraBFT protocol in order to securely replicate the state of the Libra Blockchain. We call *SMR module* the software implementation of a LibraBFT node run by each validator. From here on, we refer to participants of LibraBFT as validator nodes, or simply nodes.
- Commands sent to the SMR module are sequences of *Libra transactions*. From the point of view of the SMR module, commands and execution states are opaque data structures. The SMR module delegates the execution of commands entirely to the execution module of Libra (see possible APIs in Appendix A.1). We read the `epoch_id` from the execution state. (Recall that the current epoch stops when a change to `epoch_id` is committed.)
- Importantly, the SMR module also delegates to the rest of the system the computation of voting rights within a given epoch. This is done using the same callbacks (Appendix A.1) to the execution layer as the ones managing epochs. For better flexibility and transparency, we expect this logic to be written in Move [38], the language for programmable transactions in Libra.
- Every time a command needs to be executed, the execution engine is given a time value meant for Move smart contracts. This value is guaranteed to be consistent across every SMR node that executes the same command.
- Execution states seen by the SMR module need not be the actual blockchain data. In practice, what we call "execution state" in this report is a lightweight data structure (e.g., a hash value) that refers to a concrete execution state stored in the local storage of a validator. Every command that is committed must be executed locally at least once by every validator. In the future, LibraBFT may include additional mechanisms for a validator to synchronize with the local storage of another validator corresponding to a recent execution state.

3.2. Libra Clients

The design of LibraBFT is mostly independent of how validator nodes interact with the clients of the Libra system. However, we can make the following observations:

- Transactions submitted by the clients of Libra are first shared between validator nodes using a mempool protocol. Consensus leaders pull transactions from the mempool when they need to make a proposal.
- To authenticate the state of blockchain with respect to Libra clients, during the protocol, LibraBFT nodes sign short *commitments* to vouch that a particular execution state is being committed. This results in cryptographic *commit certificates* that are verifiable independently from the details of the consensus protocol of LibraBFT, provided that Libra clients know the set of validator keys for the corresponding epoch. We describe how commitments are created together with consensus data in section (Section 4.1).
- Regarding this last assumption, for now, we will consider that Libra clients can learn the set of validator keys from conventional interactions with one or several trusted validators. In the future, we will provide a security protocol for this purpose.

3.3. Security

Blockchain applications also require additional security considerations:

- Participants to the protocol should be able to cap the amount of resources (e.g., CPU, memory, storage, etc.) that they allocate to other nodes to ensure practical liveness despite Byzantine behaviors. We will see in the next section (Section 4) that our data-communication layer provides mechanisms to let receivers control how much data they consume (aka *back pressure*). A more thorough analysis is left for future versions of this report.
- The economic incentives of rational nodes should be aligned with the security and the performance of the SMR protocol. We sketch low-level mechanisms enabling reward and punishment in section (Section 9).
- Leaders can be subject to targeted denial-of-service attacks. Our pacemaker specifications (Section 7) sketches how to introduce a verifiable random function (VRF) [39] to assign leaders to round numbers in a less predictable way. In the future, we may also influence leader selection in order to select robust leaders more often. Protection of nodes at the system level is out of the scope of this report.

Note on Fairness. Besides safety and liveness, another abstract property often discussed in SMR systems is *fairness*. This notion is traditionally defined as the fact that every valid command submitted by an honest node is eventually committed. Yet, this classic definition is less relevant to a blockchain application such as Libra, where transactions go through a shared mempool first and are subject to auctions on transaction fees. We leave the discussion on fairness for future work.

4. Consensus Data and Networking

In this section, we introduce the core data types of LibraBFT, called *records* and discuss the communication framework used to synchronize node states over the network.

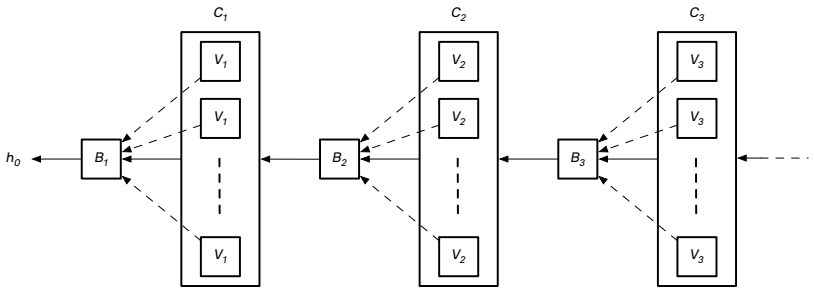

Figure 2: A chain of records in LibraBFT

4.1. Records

The core state of a LibraBFT node consists of a set of *records*. We define four kinds of records:

- *blocks*, proposed by leaders at a given *round* number and containing commands to execute.
- *votes*, by which a node votes for a block and its execution state.
- *quorum certificates* (*QC*s), which hold a quorum of votes for a given block and its execution state — and optionally a commitment meant for Libra clients.
- *timeouts*, by which a node certifies that its current round has reached a timeout.

Precise data structures in Rust are provided in the next paragraph. Most importantly:

- Records are signed by their authors.
- Blocks are *chained* (Figure 2): they must include the hash of the QC of a block at a lower round — or at the beginning of an epoch, a fixed hash value $h_{\texttt{init}}$ set during the initialization of the epoch.
- Votes and QCs include the hash of the block and the execution state that are the objects of the votes.

A QC is created by gathering enough votes to form a quorum (Section 2.3) in favor of the same execution state, according to the voting rights of the current epoch. The fact that quorum certificates are signed by their author is not essential to the protocol: this is meant only to limit the influence of the next leader(s) regarding voter rewards (see Section 9 on economic incentives).

Votes and QCs include an optional commitment value prepared for Libra clients. When a voter detects that gathering a QC on the voted block will trigger a commit for an earlier block in the chain, it must populate the field `commitment` with the execution state of that earlier block (see also the commit rule in Section 5.3 below). In this way, a QC with a non-empty field `commitment` acts as *commit certificate* — that is, a short cryptographic proof that a particular state was committed. Since we included the epoch identifier in the QC, such a commit certificate can be verified in isolation as long as one knows the set of validators for this epoch.

We include a redundant round number in votes and QCs for technical reasons related to data-synchronization messages (Appendix A.3).

When necessary, we will distinguish *network records* — records that have just been received from the network — from *verified records*, which have been thoroughly verified to ensure strong invariants defined in the next section (Section 4.2). However, we generally use *records* for *verified records* when the context is clear.

Importantly, invariants enforced by record verification (e.g., chaining rules) and the initial conditions guarantee that the records of a given epoch form a tree — with the exception of timeouts, which are

not chained. For each node, the data structure holding the records of an epoch is called a *record store*. We say that a node *knows* a record if it is present in the record store of its current epoch. We sketch the possible interfaces of the record store object in Appendix A.2. We will make explicit when nodes may delete (*clean up*) records from their record stores to minimize storage as part of the description of the protocol in Section 5.7.

Data structures in Rust. We assume the following primitive data types:

- `EpochId` (an integer).
- `Round` (an integer).
- `NodeTime` (the system time of a node).
- `BlockHash` and `QuorumCertificateHash` (hash values).
- `Author` (identifier of a consensus node).
- `Signature` (a digital signature).

The network records of LibraBFT are specified using Rust syntax in Table 1.

Hashing and Signing. We assume that the data fields of records can be hashed to produce *deterministic* hashing values. By deterministic hashes, we mean that the hashes of two data structures should be equal if and only if the content of the data structures are equal (also see Section 2.4 on cryptographic assumptions).

The hashing of records should include a type-related tag followed by all the fields in the record, except the field `signature`. Signatures of records apply to their hashing value.

The signatures in the vector of votes of a QC are copied from the original `Vote` records that were selected by the author of the QC.

4.2. Verification of Network Records

At the beginning of an epoch, consensus nodes agree on an initial value $h_{\texttt{init}}$ of type `QuorumCertificateHash`. For example, we may define $h_{\texttt{init}} = \texttt{hash}(\texttt{seed} \mid\mid \texttt{epoch\_id})$ for some fixed value `seed`.

Every consensus node sequentially verifies all the records that it receives from the network:

- All signatures should be valid signatures from a node of the current epoch.
- `BlockHash` values should refer to previously verified blocks.
- `QuorumCertificateHash` values should refer to verified quorum certificates or the initial hash $h_{\texttt{init}}$.
- Rounds should be strictly increasing for successive blocks in a chain of blocks and quorum certificates. Round numbers in proposed blocks restart at round 1 at every epoch.
- The author of a QC should be the author of the previous block.
- Epoch identifiers in timeouts, votes, and QCs must match the current epoch.
- Round values in votes and QCs must match the round of the certified block.
- The commitment value in a vote or in quorum certificate should be consistent with the commit rule (Section 5.3). If present, it should be signed by the same set of authors as the certified block.
- Network records that fail to verify should be skipped.

Given the constraints on hashes of blocks and QCs, except for timeouts, the verified records known to a node form a *tree* whose root is the value $h_{\texttt{init}}$ (Lemma S1).

```
/// A record read from the network.
enum Record {
    /// Proposed block, containing a command, e.g. a set of Libra transactions.
    Block(Block),
    /// A single vote on a proposed block and its execution state.
    Vote(Vote),
    /// A quorum of votes related to a given block and execution state.
    QuorumCertificate(QuorumCertificate),
    /// A signal that a particular round of an epoch has reached a timeout.
    Timeout(Timeout),
}

struct Block {
    /// User-defined command to execute in the state machine.
    command: Command,
    /// Time proposed for command execution.
    time: NodeTime,
    /// Hash of the quorum certificate of the previous block.
    previous_quorum_certificate_hash: QuorumCertificateHash,
    /// Number used to identify repeated attempts to propose a block.
    round: Round,
    /// Creator of the block.
    author: Author,
    /// Signs the hash of the block, that is, all the fields above.
    signature: Signature,
}

struct Vote {
    /// The current epoch.
    epoch_id: EpochId,
    /// The round of the voted block.
    round: Round,
    /// Hash of the certified block.
    certified_block_hash: BlockHash,
    /// Execution state.
    state: State,
    /// Execution state of the ancestor block (if any) that will match
    /// the commit rule when a QC is formed at this round.
    commitment: Option<State>,
    /// Creator of the vote.
    author: Author,
    /// Signs the hash of the vote, that is, all the fields above.
    signature: Signature,
}

struct QuorumCertificate {
    /// The current epoch.
    epoch_id: EpochId,
    /// The round of the certified block.
    round: Round,
    /// Hash of the certified block.
    certified_block_hash: BlockHash,
    /// Execution state
    state: State,
    /// Execution state of the ancestor block (if any) that matches
    /// the commit rule thanks to this QC.
    commitment: Option<State>,
    /// A collections of votes sharing the fields above.
    votes: Vec<(Author, Signature)>,
    /// The leader who proposed the certified block should also sign the QC.
    author: Author,
    /// Signs the hash of the QC, that is, all the fields above.
    signature: Signature,
}

struct Timeout {
    /// The current epoch.
    epoch_id: EpochId,
    /// The round that has timed out.
    round: Round,
    /// Creator of the timeout object.
    author: Author,
    /// Signs the hash of the timeout, that is, all the fields above.
    signature: Signature,
}
```

Table 1: Network records in LibraBFT

4.3. Communication Framework

In LibraBFT, the communication framework builds a peer-to-peer overlay for the reliable dissemination of protocol records (Section 4.1) among the validators. The framework API consists of two primitives actions: (i) *send*, where the node updates a peer with the records it has; (ii) *broadcast*, where the node disseminates updates to all its peers.

In order to provide reliable delivery guarantees to honest nodes, LibraBFT builds gossip overlay on top of a point-to-point synchronization protocol. Briefly, a node that wishes to broadcast sends its data to a random subset of at least K nodes, for some fixed value K ($0 < K \leq N$). Receiving nodes reshare relevant data in the same way.

The exact nature of the "relevant" data sent and reshared during network actions is part of the description of the LibraBFT protocol (Section 7.11). In the first reading, one may simply consider that a node reshares every valid record that it knows every time its record store is updated.

Resharing data is important to make the broadcast action *reliable* [24] in the following sense: if an honest node receives (relevant) data, then — with high probability — every other honest node will know about these data shortly after. Importantly, this holds even if the origin of the data is a malicious node.

Randomized gossip provides the following broadcast guarantee:

> **(probabilistic-reliable-broadcast)** After GST, if an honest node receives or possesses data that requires gossiping, then — with high probability — before an (unknown) time delay $\delta_G > 0$, every other honest node will have received these data.

We interpret this requirement in a broad sense that allows data to be updated along the way. We also allow several senders to initiate the gossiping of the same data in parallel during an interval of time $[t_1; t_2]$ and assume that the data is received by all honest nodes by time $t_2 + \delta_G$.

Note on choosing the fan-out factor. The fan-out factor K is generally chosen to be much smaller than the number of nodes, as a trade-off between networking delay and scalability. We leave for future work how to choose a value K so that the requirement above follows from the classic assumption (eventually-synchronous-network) on single messages after GST. For now, we may simply choose $K = N$ to include every node, and let $\delta_G = \delta_M$.

4.4. Data Synchronization

Because of resharing and the possibility of crashes, a naive approach, where senders push their records directly to receivers, would lead to receiving and retransmitting the same data many times. In general, we wish to let senders initiate communication but give more control to receivers on how they consume data. In LibraBFT, this is addressed by introducing an exchange protocol called *data synchronization*.

Sending data from one sender to other nodes is done in multiple steps:

- Make the new data available (i.e., passively publish it) as part of the *data-synchronization service* of the sender.
- Send a notification, called a `DataSyncNotification` message, to each receiver according to the nature of the communication: a single receiver for point-to-point communication, a random subset otherwise.

- Let receivers connect back to the sender with `DataSyncRequest` message and retrieve data contained in a `DataSyncResponse` message.

When the exchange is completed, a receiver should validate received data immediately, then make all valid and relevant data available as a server. As mentioned earlier, new data may require notifying further nodes to complete the reliable broadcast.

4.5. Runtime Environment

For the purpose of specifying and simulating LibraBFT, we abstract away details about processes, networking, timers, and, generally, the operating systems of nodes under the generic term *runtime environment*. We will specify the behavior of LibraBFT nodes as the combination of a private local state and a small number of algorithms called *handlers*. A handler typically mutates the local state of the current node and returns a value. Specifications will require the runtime environment to call handlers at specific times and interpret returned values right away.

4.6. Data-Synchronization Handlers

We now specify the handlers for data synchronization. We require the runtime environment to carry the messages sketched in Section 4.4 (namely `DataSyncNotification`, `DataSyncRequest`, and `DataSyncResponse`) to their intended recipient in an authenticated channel, at a speed depending on the current network conditions. We create three corresponding handlers to be called when a message is received and returning a possible answer. An additional handler `create_notification` is used when the main handler of node (Section 5.6) requests the environment to send a notification to specific senders.

Interface in Rust. We express the data-synchronization handlers as a Rust trait as follows:

```
trait DataSyncNode {
    /// Sender role: what to send to initiate a data-synchronization exchange.
    /// We only include our current vote when notifying a proposer.
    fn create_notification(&self, include_vote: bool) -> DataSyncNotification;
    /// Sender role: handle a request from a receiver.
    fn handle_request(&mut self, request: DataSyncRequest) -> DataSyncResponse;
    /// Receiver role: accept or refuse a notification from an authenticated sender.
    fn handle_notification(
        &mut self,
        authenticated_sender: Author,
        notification: DataSyncNotification,
        smr_context: &mut SMRContext,
    ) -> Option<DataSyncRequest>;
    /// Receiver role: receive data from an authenticated sender.
    fn handle_response(
        &mut self,
        authenticated_sender: Author,
        response: DataSyncResponse,
        smr_context: &mut SMRContext,
    );
}
```

Data-synchronization handlers continuously query and update the record store of a node, independently from the main handler of the protocol, which will be presented in Section 5. Possible definitions for the three message types are given in Appendix A.3.

4.7. Mathematical Notations

We have seen that records that fail to be verified are rejected from receiving nodes. Unless mentioned otherwise, all records considered from now on are verified records.

We use the letter α to denote a node of the protocol. We write $\texttt{record\_store}(\alpha)$ for the record store of α at a given time. We use the symbol || to denote the concatenation of bit strings.

We introduce the following notations regarding records:

- We use the letter B to denote block values; C to denote quorum certificates; V for votes; T for timeouts; and, finally, R to denote either blocks or certificates.
- We use h, h_1, etc., to denote hash values of type `QuorumCertificateHash` or `BlockHash`. We use letters n, n_1, etc., for rounds.
- We write $\texttt{round}(B)$ for the field `round` of a block and, more generally, $\texttt{foo}(R)$ for any field `foo` of a record R.
- If $h = \texttt{certified\_block\_hash}(C)$, we write $h \leftarrow C$. Similarly, we write $h \leftarrow V$ in case of a single vote V. If $h = \texttt{previous\_quorum\_certificate\_hash}(B)$, we write $h \leftarrow B$.
- More generally, we see $\leftarrow$ as a relation between hashes, blocks, votes, and quorum certificates. We may write $B \leftarrow C$ instead of $\texttt{hash}(B) \leftarrow C$, $B \leftarrow V$ for $\texttt{hash}(B) \leftarrow V$, and $C \leftarrow B$ instead of $\texttt{hash}(C) \leftarrow B$.
- Finally, we write $\leftarrow^*$ for the transitive and reflexive closure of $\leftarrow$, that is: $R_0 \leftarrow^* R_n$ if and only if $R_0 \leftarrow R_1 ... \leftarrow R_n$, $n \geq 0$.

5. The LibraBFT Protocol

5.1. Overview of the Protocol

Each consensus node α maintains a local tree of records for the current epoch, previously noted $\texttt{record\_store}(\alpha)$. The initial root of the tree, a QC hash noted $h_{\texttt{init}}$, is agreed upon as part of the setup for the consensus epoch. Each branch in the tree is a chain of records, alternating between blocks B_i and quorum certificates C_i. Formally, such as a chain is denoted: $h_{\texttt{init}} \leftarrow B_1 \leftarrow C_1 ... \leftarrow B_n[\leftarrow C_n]$.

When a node acts as a *leader* (Figure 3), it must propose a new block of transactions B_{n+1}, usually extending the tail quorum certificate C_n of (one of) its longest branch(es) (❶). Assuming that the proposal B_{n+1} is successfully broadcast, honest nodes will verify the data, execute the new block, and send back a vote to the leader (❷). In the absence of execution bugs, honest nodes should agree on the execution state after B_{n+1}. Upon receiving enough votes agreeing with this execution state, the proposer will create a quorum certificate C_{n+1} for this block and broadcast it (❸). The chain length has now increased by one: $h_{\texttt{init}} \leftarrow B_1 \leftarrow C_1 ... \leftarrow B_{n+1} \leftarrow C_{n+1}$. At this point, the leader is considered done, and another leader is expected to extend the tree with a new proposal.

Due to network delays and malicious nodes, honest nodes may not always agree on the "best" branch to extend and for which blocks to vote. Under BFT assumption (Section 2.3), the *voting constraints* observed by honest nodes guarantee that when a branch grows enough to include a block B that satisfies the *commit rule*, B and its predecessors cannot be challenged by conflicting proposals anymore. These blocks are thus *committed* in order to advance the replicated state machine.

To guarantee progress despite malicious nodes or unresponsive leaders, each proposal includes a *round* number. A round will time out after a certain time. When the next round becomes *active*, a new

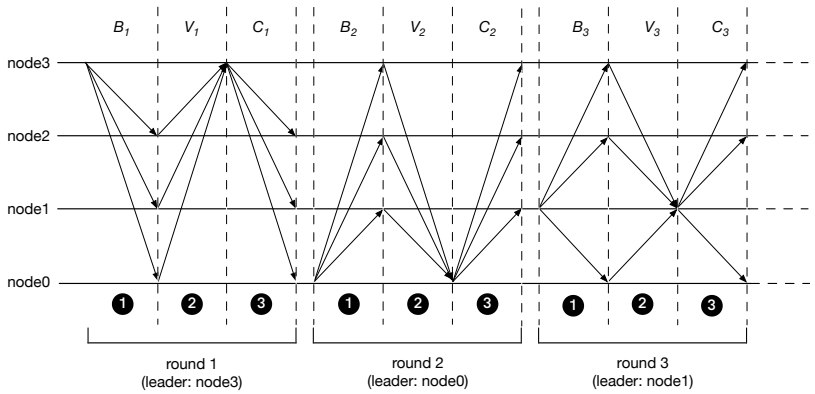

Figure 3: Overview of the LibraBFT protocol (simplified, excluding round synchronization)

leader is expected to propose a block. The *pacemaker* abstraction (Section 7.3) aims to make honest nodes agree on a unique, active round for sufficiently long periods of time.

We can now rephrase the main goals of the LibraBFT protocol as follows:

> **(safety)** New commits always extend a chain containing all the previous commits.
>
> **(liveness)** If the network is synchronous for a sufficiently long time, eventually a new commit is produced.

Layout of the description of LibraBFT. In the rest of this section, we make precise the commit rule (Section 5.3) and the voting constraints (Section 5.4 and Section 5.5). Then, using the communication framework described previously in Section 4, we proceed to describe the local state and the behaviors of nodes in the LibraBFT protocol (Section 5.6 and Section 5.7).

This section provides the prerequisites for the proof of safety given in Section 6. Liveness mechanisms will be presented in Section 7 and followed by the proof of liveness in Section 8.

5.2. Chains

A *k-chain* is a sequence of k blocks and k QCs:

$$B_0 \leftarrow C_0 \leftarrow ... \leftarrow B_{k-1} \leftarrow C_{k-1}$$

B_0 is called the *head* of such a chain. C_{k-1} is called the *tail*.

Recall that by definition of the notion of round for blocks, rounds must be strictly increasing along a chain: $\texttt{round}(B_i) < \texttt{round}(B_{i+1})$.

When rounds increase exactly by one — that is, $\texttt{round}(B_i) + 1 = \texttt{round}(B_{i+1})$ — we say that the chain has *contiguous rounds*.

In practice, the round numbers of a chain may fail to be contiguous for many reasons. For example, a dishonest leader may propose an invalid block, or a leader may fail to gather a quorum of votes in a timely manner because of network issues. When a quorum certificate is not produced in a round, a leader at a higher round will eventually propose a block that breaks contiguity in the chain.

5.3. Commit Rule

A block B_0 is said to *match the commit rule* of HotStuff in the record store of a node if and only if it is the head of a 3-chain with contiguous rounds, that is, there exist C_0, B_1, C_1, B_2, C_2 such that

$$B_0 \leftarrow C_0 \leftarrow B_1 \leftarrow C_1 \leftarrow B_2 \leftarrow C_2$$

and

$$\texttt{round}(B_2) = \texttt{round}(B_1) + 1 = \texttt{round}(B_0) + 2$$

When such a commit rule is observed by a node α, the blocks preceding B_0 in the record store of α, and B_0 itself becomes *committed*.

Following our previous discussion (Section 4.1) on commitments, a valid quorum certificate in the position of C_2 acts as a *commit certificate*: it must include a non-empty field value `commitment` to authenticate that the execution state $\texttt{state}(C_0)$ was committed in the current epoch. Note that if `commitment` is empty, then C_2 is not a valid record, and it should be ignored (Section 4.2).

5.4. First Voting Constraint: Increasing Round

Safety of the commit rule relies on two *voting constraints*. The first one concerns the rounds of voted blocks:

> **(increasing-round)** An honest node that voted once for B in the past may only vote for B' if $\texttt{round}(B) < \texttt{round}(B')$.

This voting constraint is important for quorum certificates (see Section 6). In practice, a node α will track the round of its latest vote in a local variable noted $\texttt{latest\_voted\_round}(\alpha)$, and only vote for a block B if $\texttt{round}(B) > \texttt{latest\_voted\_round}(\alpha)$.

5.5. Second Voting Constraint: Locked Round

The *locked round* of a node α, written $\texttt{locked\_round}(\alpha)$, is the highest round of the head of a 2-chain ever known to α, if any, and zero otherwise. In practice, we may initialize the value $\texttt{locked\_round}(\alpha)$ to 0 and update it to $\texttt{round}(B_0)$ whenever a new 2-chain $B_0 \leftarrow C_0 \leftarrow B_1 \leftarrow C_1$ such that $\texttt{round}(B_0) > \texttt{locked\_round}(\alpha)$ is found in $\texttt{record\_store}(\alpha)$.

We also define the *previous round* of a block B as follows: if there exist B' and C' such that $B' \leftarrow C' \leftarrow B$, we let $\texttt{previous\_round}(B) = \texttt{round}(B')$; otherwise, $\texttt{previous\_round}(B) = 0$.

We can now formulate our second *voting constraint*:

> **(locked-round)** An honest node α may only vote for a block B if it currently holds that $\texttt{previous\_round}(B) \geq \texttt{locked\_round}(\alpha)$.

The voting constraint (locked-round) was adapted from the most recent version of HotStuff [5]. In LibraBFT, it is simplified into a single-clause condition.

5.6. Local State of a Consensus Node and Main Handler API

We can now describe the protocol followed by a LibraBFT node in terms of a local state and a handler called by the runtime environment (Section 4.5).

Local state. As mentioned previously, the core component of the state of a node α consists of the current record store (Section 4.2) — written as `record_store`(α) — which contains all the verified records that α knows for its current epoch.

The state of node also includes a number of variables related to leader election. We group them into a special object called a *pacemaker* and describe them in Section 7.3.

Additional state variables needed by an instance of the protocol include:

- The current epoch identifier `epoch_id`(α), used to detect the end of the current epoch;
- The identifier of α as an author of records, written `local_author`(α);
- The round of the latest voted block `latest_voted_round`(α), (initial value: 0);
- The locked round `locked_round`(α), (initial value: 0);
- The identities and the active rounds of the latest nodes that synchronized with us, denoted `latest_senders` (initial value: the empty list); and
- The system time of the last broadcast `latest_broadcast`(α), (initial value: the starting time of the epoch).

The state of a node also includes an object, called *data tracker*, responsible for tracking data that the main handler has already processed, notably commits, and deciding if new data need to be reshared. We give more details about the data tracker in Section 7.11.

Finally, the state of a node includes the record stores of all the previous epochs. Those epochs are now stopped, meaning that no new records can be inserted.

Main handler API. In LibraBFT, the main handler of a consensus node consists of a single algorithm `update_node` that must be called by the runtime environment on three occasions:

- Whenever the node starts or restarts after a crash;
- Whenever a data-synchronization exchange was completed (Section 4.6); and
- Regularly, at a given time scheduled by the last run of the handler itself.

The main handler reacts to the changes observed in the record store or in the clock by returning a list of *action items* to be carried by the runtime environment. Specifically:

- The main handler may require that a new call to `update_node` be scheduled at a given time in the future;
- It may specify that a data notification should be sent to a particular leader; and
- It may ask to broadcast data notifications.

The implementation of the main handler is the core of the LibraBFT protocol. It is described in detail in Section 5.7.

Rust definitions. In Rust, the local state of a node is written as follows:

```
struct NodeState {
    /// Module dedicated to storing records for the current epoch.
    record_store: RecordStoreState,
    /// Module dedicated to leader election.
    pacemaker: PacemakerState,
    /// Current epoch.
    epoch_id: EpochId,
    /// Identity of this node.
    local_author: Author,
    /// Highest round voted so far.
    latest_voted_round: Round,
    /// Current locked round.
    locked_round: Round,
    /// Time of latest broadcast.
    latest_broadcast: NodeTime,
    /// Names and rounds of the latest senders during network communication.
```

```
    latest_senders: Vec<(Author, Round)>,
    /// Track data to which the main handler has already reacted.
    tracker: DataTracker,
    /// Record stores from previous epochs.
    past_record_stores: HashMap<EpochId, RecordStoreState>,
}
```

The main handler API, `update_node`, is written:

```
trait ConsensusNode {
    fn update_node(&mut self, clock: NodeTime, smr_context: &mut SMRContext) -> NodeUpdateActions;
}
```

This definition assumes that the runtime environment provides the following inputs:

- The current node state (`self` in Rust);
- The current system time (`clock`); and
- A context for SMR operations such as command execution (`smr_context`).

Recall that the trait `ConsensusNode` comes in addition to the trait `DataSyncNode` (Section 4.6), which provide handlers for data synchronization. The trait `SMRContext` is made precise in Appendix A.1.

Action items returned by the function `update_node` are held in the following data structure:

```
struct NodeUpdateActions {
    /// Time at which to call `update_node` again, at the latest.
    should_schedule_update: Option<NodeTime>,
    /// Whether we need to send a notification to a leader.
    should_notify_leader: Option<Author>,
    /// Whether we need to send notifications to a random subset of nodes.
    should_broadcast: bool,
}
```

5.7. Main Handler Implementation

We now describe the implementation of the main handler of a LibraBFT node in Rust (Table 2).

At a high level, the algorithm realizes the following operations:

- Run the pacemaker module and execute requested pacemaker actions, such as creating a timeout or proposing a block.
- Execute and vote for a valid proposed block, if any, while respecting the two voting constraints regarding the latest voted round (Section 5.4) and the locked round (Section 5.5).
- If the node proposed a block and if a quorum of votes for the same state was received, create a quorum certificate.
- Check for newly found commits in the record store and deliver them to the state-machine replication context.
- Check for a commit that would terminate the current epoch, and start a new one if needed.
- Decide if the node should reshare (i.e., gossip) its data.

This algorithm relies on the record store of a node for computing the round of the highest head of a known 2-chain (`highest_2chain_head_round`), the head of the highest commit rule (`highest_committed_round`), and the tail QC of a commit rule (`commit_certificate`).

The method `chain_between_quorum_certificates` takes two rounds m and n ($m \leq n$) as inputs. If the record store contains a chain $B_0 \leftarrow C_0 \leftarrow B_1 \leftarrow C_1 \leftarrow \ldots \leftarrow B_k \leftarrow C_k$ with $\texttt{round}(B_0) = m$ and $\texttt{round}(B_k) = n$, then it will return an iterator on the QCs $C_1, \ldots, C_k$, in this order. (See Appendix A.2 for detailed interfaces.)

```
impl ConsensusNode for NodeState {
    fn update_node(&mut self, clock: NodeTime, smr_context: &mut SMRContext) -> NodeUpdateActions {
        // Update pacemaker state and process pacemaker actions (e.g., creating a timeout, proposing a block).
        let latest_senders = self.read_and_reset_latest_senders();
        let pacemaker_actions = self.pacemaker.update_pacemaker(
            self.local_author,
            &self.record_store,
            self.latest_broadcast,
            latest_senders,
            clock,
        );
        let mut actions = self.process_pacemaker_actions(pacemaker_actions, smr_context);
        // Update locked round.
        self.locked_round = std::cmp::max(self.locked_round, self.record_store.highest_2chain_head_round());
        // Vote on a valid proposal block designated by the pacemaker, if any.
        if let Some((block_hash, block_round, proposer)) = self.record_store.proposed_block(&self.pacemaker) {
            // Enforce voting constraints.
            if block_round > self.latest_voted_round
                && self.record_store.previous_round(block_hash) >= self.locked_round
            {
                // Update latest voted round.
                self.latest_voted_round = block_round;
                // Try to execute the command contained the a block and create a vote.
                if self.record_store.create_vote(self.local_author, block_hash, smr_context) {
                    // Ask that we reshare the proposal.
                    actions.should_broadcast = true;
                    // Ask to notify and send our vote to the author of the block.
                    actions.should_notify_leader = Some(proposer);
                }
            }
        }
        // Check if our last proposal has reached a quorum of votes and create a QC.
        if self.record_store.check_for_new_quorum_certificate(self.local_author, smr_context) {
            // The new QC may cause a change in the pacemaker state: schedule a new run of this handler now.
            actions.should_schedule_update = Some(clock);
        }
        // Check for new commits and verify if we should start a new epoch.
        for commit_qc in self
            .record_store
            .chain_between_quorum_certificates(
                self.tracker.highest_committed_round,
                self.record_store.highest_committed_round(),
            )
            .cloned()
        {
            // Deliver the new committed state, together with a short certificate (if any).
            smr_context.commit(&commit_qc.state, self.record_store.commit_certificate(&commit_qc));
            // If the current epoch ended..
            let epoch_id = smr_context.read_epoch_id(&commit_qc.state);
            if self.epoch_id != epoch_id {
                // .. create a new record store and switch to the new epoch.
                self.start_new_epoch(epoch_id, commit_qc, smr_context);
                // .. stop delivering commits after an epoch change.
                break;
            }
        }
        // Update the data tracker and ask that we reshare data if needed.
        if self.tracker.update_and_decide_resharing(self.epoch_id, &self.record_store) {
            actions.should_broadcast = true;
        }
        // Return desired node actions to environment.
        actions
    }
}
```

Table 2: Main handler of a LibraBFT node

The main handler also uses additional interfaces related to liveness and described in later sections:

- The `Pacemaker` trait provides a function `update_pacemaker` to control leader election, timeouts, and proposals; the returned action items are processed by a method `process_pacemaker_actions`; and the method `proposed_block` of `RecordStore` also uses the pacemaker to select an *active proposal* that a node can vote for, if any (Section 7.3).
- The `DataTracker` object provides the latest commit processed so far, as well as a method `update_and_decide_resharing` to update the latest commit value (among others) and control gossiping (Section 7.11).

Epoch changes. As soon as a node delivers a commit QC that ends the current epoch, it stops delivering commits for this epoch, archives its current record store, and creates a record store for the new epoch. Nodes must keep the record stores of the previous epochs to ensure liveness: during data synchronization, a node must be able to follow the chain of commits and execute commands up to the latest commit rule of the latest epoch of any sender (see also Section 3.1 and Section 7.11).

The chain of commits between the block containing the epoch changes and the block triggering the commit rule may be arbitrarily long depending on network conditions. To avoid persisting data that will not be committed, we may require proposers to propose only empty commands once an epoch change is detected on a branch.

6. Proof of Safety

In the proof of safety, we consider the set of all records ever seen by honest nodes in the current epoch and prove that committed blocks must form a linear chain $h_{\texttt{init}} \leftarrow B_1 \leftarrow C_1 \leftarrow B_2 \ldots \leftarrow B_n$, starting from the initial QC hash $h_{\texttt{init}}$ of the epoch.

6.1. Preliminaries

We start by recalling the proof of the classical BFT lemma:

Lemma B1: Under BFT assumption, for every two quorums of nodes in the same epoch, there exists an honest node that belongs to both quorums.

Proof: Let $M_i \geq N - f$ $(i = 1, 2)$ be the combined voting power of each quorum. The voting powers M_i' of each quorum, excluding Byzantine nodes, satisfies $M_i' \geq M_i - f \geq N - 2f$. We note that if the two sets were disjoint, the voting power of the union $M_1' + M_2' \geq 2N - 4f > N - f$ would exceed the voting power of all honest nodes. Therefore, there exists an honest node in both quorums. □

Next, we prove two new lemmas. The first one concerns the chaining of records.

Lemma S1: For any records R, R_0, R_1, R_2:

- $h_{\texttt{init}} \leftarrow^* R$;
- If $R_0 \leftarrow R_2$ and $R_1 \leftarrow R_2$ then $R_0 = R_1$; and
- If $R_0 \leftarrow^* R_2$, $R_1 \leftarrow^* R_2$ and $\texttt{round}(R_0) < \texttt{round}(R_1)$ then $R_0 \leftarrow^* R_1$.

PROOF:

- By definition of verified records (Section 4.2).
- By definition of chaining, assuming that hashing is perfectly collision resistant.
- By induction on the derivation $R_1 \leftarrow^* R_2$, using the previous item and the fact that rounds cannot decrease in a chain. □

The second lemma concerns the first voting rule and the notion of quorum certificates.

Lemma S2: Consider two blocks with QCs: $B \leftarrow C$ and $B' \leftarrow C'$. Under BFT assumption, if $\texttt{round}(B) = \texttt{round}(B')$, then $B = B'$ and $\texttt{state}(C) = \texttt{state}(C')$.

In particular, for every $k > 0$, there is a unique block that has the highest round amongst the heads of k-chains known to a node.

PROOF: Under BFT assumption, there must exist an honest node that voted both for the winning proposal $\texttt{state}(C)$ in C and for $\texttt{state}(C')$ in C'. By the voting rule (increasing-round), we must have $B = B'$ and $\texttt{state}(C) = \texttt{state}(C')$. □

6.2. Main Safety Argument

We say that two (distinct) records R, R' are *conflicting* when neither $R \leftarrow^* R'$ nor $R' \leftarrow^* R$.

Lemma S3: Assume a 3-chain starting at round n_0 and ending at round n_2. For every certified block $B \leftarrow C$, such that $\texttt{round}(B) > n_2$, under BFT assumption, we have $\texttt{previous\_round}(B) \geq n_0$.

PROOF: Let $B_0 \leftarrow C_0 \leftarrow B_1 \leftarrow C_1 \leftarrow B_2 \leftarrow C_2$ be a 3-chain starting at round $n_0 = \texttt{round}(B_0)$ and ending at round $n_2 = \texttt{round}(B_2)$.

Under BFT assumption, there exists an honest node α whose vote is included both in C_2 (voting for B_2) and C (voting for B). Since $\texttt{round}(B) > n_2$, by the voting rule (increasing-round), α must have voted for B_2 first. At that time, α had already seen the 2-chain starting with B_0. Since the locked round never decreases, its locked round was at least $\texttt{round}(B_0) = n_0$. At the later time of voting for B, the locked round of α was again at least n_0. Therefore, the voting rule (locked-round) implies that $\texttt{previous\_round}(B) \geq n_0$. □

Proposition S4: Assume a 3-chain with contiguous rounds starting with a block B_0 at round n_0. For every certified block $B \leftarrow C$, such that $\texttt{round}(B) \geq n_0$, under BFT assumption, we have $B_0 \leftarrow^* B$.

PROOF: By induction on $\texttt{round}(B) \geq n_0$. Let $B_0 \leftarrow C_0 \leftarrow B_1 \leftarrow C_1 \leftarrow B_2 \leftarrow C_2$ be a 3-chain starting with B_0 and with contiguous rounds: $\texttt{round}(B_0)+2 = \texttt{round}(B_1)+1 = \texttt{round}(B_2) = n_0+2$.

If $\texttt{round}(B) \leq n_0 + 2$, then $\texttt{round}(B)$ is one of the values n_0, $n_0 + 1$, $n_0 + 2$. By Lemma S2, B is one of the values B_0, B_1, B_2; therefore, $B_0 \leftarrow^* B$.

Otherwise, assume $\texttt{round}(B) > n_0 + 2$, that is, $\texttt{round}(B) > \texttt{round}(B_2)$. By Lemma S3, we have $\texttt{previous\_round}(B) \geq n_0$. Since $n_0 = \texttt{round}(B_0) > 0$, this means there exists a chain $B_3 \leftarrow C_3 \leftarrow B$ such that $\texttt{round}(B_3) \geq n_0$. Since $\texttt{round}(B_3) \geq n_0$ and $\texttt{round}(B_3) < \texttt{round}(B)$, we may apply the induction hypothesis on B_3 to deduce that $B_0 \leftarrow^* B_3$. Therefore, $B_0 \leftarrow^* B_3 \leftarrow^* B$ concludes the proof. □

Theorem S5 (Safety): Under BFT assumption, two blocks that match the commit rule cannot be conflicting.

PROOF: Consider the commit rules of two blocks B_0 and B'_0 that match the commit rule: $B_0 \leftarrow C_0 \leftarrow B_1 \leftarrow C_1 \leftarrow B_2 \leftarrow C_2$ and $B'_0 \leftarrow C'_0 \leftarrow B'_1 \leftarrow C'_1 \leftarrow B'_2 \leftarrow C'_2$ (with contiguous rounds in both cases). Without loss of generality, we may assume that $\texttt{round}(B'_0) \geq \texttt{round}(B_0)$. By Proposition S4, this implies $B_0 \leftarrow^* B'_0$. □

Corollary S6: Under BFT assumption, the set of all commits seen by any honest node since the beginning of the current epoch form a linear chain $h_{\texttt{init}} \leftarrow B_1 \leftarrow C_1 \leftarrow B_2 \ldots \leftarrow B_n$.

PROOF: Using Theorem S5, by induction on the number of commits. □

7. Liveness Mechanisms of LibraBFT

LibraBFT follows the example of HotStuff [5] and delegates leader election to a special module called a *pacemaker*. We now describe the pacemaker of LibraBFT in detail, as well as the policies for resharing data and cleaning the record store. These mechanisms are all crucial for liveness, as we will see in the proofs of Section 8.

7.1. Timeout certificates.

We define a *Timeout Certificate* (TC) as a set of timeout objects at the same round n, such that the combined voting power of timeout authors exceeds f. The round of a timeout certificate is the common round value n.

7.2. Overview of the Pacemaker

Given the latest committed block and its QC, noted $B_c \leftarrow C_c$, we assign a *leader* and a *maximum duration* to every round number $n > \texttt{round}(B_c) + 2$. A consensus node enters a round n whenever it receives a QC at round $n-1$, or enough timeouts to form a TC at round $n-1$, whichever comes first. The round n is then considered *active* until the node enters round $n+1$. While a node has active round n, it may only vote for a block at round n authored by the leader of round n.

To achieve liveness despite malicious or unresponsive leaders, nodes start a timer when they enter a round and verify that the elapsed time has not passed the current maximum duration of the round. New timeout objects are gossiped immediately. Once enough timeouts have been shared, a node will observe a TC and change their active round — that is, unless the node learned a QC first, which also updates the active round. The maximum duration of an active round is subject to be updated if the latest committed block changes while the round is active.

Importantly, when a node enters a round that makes it a leader, it must wait for a quorum of nodes to confirm that they entered the round as well before proposing a block. This is important so that the proposed block can provably meet the second voting constraint computed by a quorum of honest nodes (Section 8.3).

Finally, during periods of asynchrony, some nodes may become unaware of the latest committed block. To recover from this situation after GST, we make sure that nodes broadcast their states at least once per period of time $I > 0$.

7.3. Pacemaker State and Update API

We specify the pacemaker module at a high level in the same way as the node itself (Section 5.6) in terms of local state and an update API.

Visible pacemaker state. The pacemaker module is in charge of driving leader election. As such, it exposes two important state values to the other components of a LibraBFT node:

- The current active round, denoted `active_round`(α), (initial value: 0);
- The leader of the current round, written `active_leader`(α), (initial value: $\perp$).

These two values are accessed notably by the function `proposed_block` of the record store, which was used earlier in the main handler `update_node` (Section 5.7). Specifically, this function ensures that the block B that we may vote for satisfies `round`(B) = `active_round`(α) and `author`(B) = `active_leader`(α).

The rest of the pacemaker state will be described in Section 7.9.

Pacemaker update API. We expect a pacemaker module to provide a method `update_pacemaker`, meant to be called by the higher-level method `update_node`. This method should refresh the state of the pacemaker in the function of the input arguments and return a list of action items for the main handler of a node to process.

Action items returned by `update_pacemaker` are similar to the action items returned by `update_node`, augmented with the two internal action items:

- The pacemaker may instruct the node to create a timeout object for the given round.
- The pacemaker may require that the node act as the leader and propose a new block.

Rust definitions. The Rust trait for a pacemaker module is written as follows:

```
trait Pacemaker {
    /// Update our state from the given data and return some action items.
    fn update_pacemaker(
        &mut self,
        // Identity of this node
        local_author: Author,
        // Tree of records
        record_store: &RecordStore,
        // Local time of the latest broadcast by us
        latest_broadcast: NodeTime,
        // Known active rounds of recent senders
        latest_senders: Vec<(Author, Round)>,
        // Current local time
        clock: NodeTime,
    ) -> PacemakerUpdateActions;

    /// Current active round and current leader.
    fn active_round(&self) -> Round;
    fn active_leader(&self) -> Option<Author>;
}
```

The parameters passed to `update_pacemaker` were described previously in the main handler `update_node` (Table 2).

The pacemaker action items returned by `update_pacemaker` can be described as follows:

```
struct PacemakerUpdateActions {
    /// Time at which to call `update_pacemaker` again, at the latest.
    should_schedule_update: Option<NodeTime>,
    /// Whether we should create a timeout object for the given round.
    should_create_timeout: Option<Round>,
    /// Whether we need to send our records to the given next leader.
    should_notify_leader: Option<Author>,
    /// Whether we need to broadcast our records.
```

```
    should_broadcast: bool,
    /// Whether to propose a block and on top of which QC hash.
    should_propose_block: Option<QuorumCertificateHash>,
}
```

These action items are interpreted by `update_node` and turned into node action items as follows:

```
impl NodeState {
    fn process_pacemaker_actions(
        &mut self,
        pacemaker_actions: PacemakerUpdateActions,
        smr_context: &mut SMRContext,
    ) -> NodeUpdateActions {
        let mut actions = NodeUpdateActions::new();
        actions.should_schedule_update = pacemaker_actions.should_schedule_update;
        actions.should_broadcast = pacemaker_actions.should_broadcast;
        actions.should_notify_leader = pacemaker_actions.should_notify_leader;
        if let Some(round) = pacemaker_actions.should_create_timeout {
            self.record_store.create_timeout(self.local_author, round, smr_context);
        }
        if let Some(previous_qc_hash) = pacemaker_actions.should_propose_block {
            self.record_store.propose_block(
                self.local_author,
                previous_qc_hash,
                self.latest_broadcast,
                smr_context,
            );
        }
        actions
    }
}
```

7.4. Equivalence of Quorum Certificates

To define leaders and maximum durations rigorously, including at the beginning of an epoch, we must take some precautions and introduce the notion of *QC equivalence*.

Two QC hashes h and h' are said to be *equivalent*, noted $h \approx h'$, if and only if one of the two conditions is fulfilled:

1) $h = h'$; or
2) There exist two quorum certificates C and C' and a block B, such as $h = \texttt{hash}(C)$, $h' = \texttt{hash}(C')$, $B \leftarrow C$, $B \leftarrow C'$, and $\texttt{state}(C) = \texttt{state}(C')$.

The first condition matters only for initial hashes. In the second case, we say that the quorum certificates C and C' are *equivalent* and write $C \approx C'$.

The reason behind this definition is that a block may be known to have a valid QC, for example, $B_0 \leftarrow C_0$, but consensus nodes may temporarily disagree on C_0. Indeed, although we required C_0 to be signed by the author of B_0, a dishonest proposer could select and aggregate votes in different ways and broadcast several variants of C_0. We have seen that under BFT assumption, all variants must be equivalent (Lemma S2) in the sense defined above.

Notations: In the following, we will h_c for the hash of the QC of the latest committed block, if any; otherwise, the initial hash of the epoch. We will require that formulas for leaders and maximum durations depend on h_c up to equivalence — they may depend on the state and the committed block, but not the voters.

When we use h_c as an input for a function, we will rely on the fact that a record store is available locally to each node and can resolve the latest commit hash h_c into actual data.

7.5. Prerequisite: Assigning Leaders to Rounds

Given a suitable record store, a QC hash h_c, and a round n, we assume an algorithm $\texttt{leader}(h_c, n)$ that returns an author in a *fair* way, meaning that all sequences of k authors ($k > 0$) are equally frequent. As explained above (Section 7.4), we also require that $h_c \approx h'_c$ implies $\texttt{leader}(h_c, n) = \texttt{leader}(h'_c, n)$.

Assuming equal voting rights, the simplest approach is to let $\texttt{leader}(h_c, n) = \texttt{author}(\texttt{hash}(n) \mod N)$, where N is the number of nodes. However, this lets anyone predict leaders for a long time in advance. This is problematic as it facilitates the preparation of targeted attacks on leaders.

We also note that depending on h_c in a naive way is not possible because of grinding attacks — a leader at round n could try to select transactions, or votes, so that $\texttt{leader}(h_c, n')$ ($n' > n+2$) points to a particular node once $n = \texttt{round}(h_c)$.

To mitigate both risks, we intend to use a verifiable random function (VRF) [39] in the future. If the certified block under the QC hash h_c contains some seed $s = \texttt{VRF}_{\texttt{author}(h_c)}(\texttt{epoch\_id} \,||\, \texttt{round}(h_c))$, then we may define $\texttt{leader}(h_c, n) = \texttt{author}(\texttt{PRF}_s(n) \mod N)$ where PRF stands for the implementation of a pseudo-random function.

7.6. Prerequisite: Minimum Broadcast Interval

We assume a time delay $I > 0$ fixed as part of the protocol. The shorter the delay, the more responsive to a newly synchronous network we will be.

7.7. Prerequisite: Delay Coefficients

We assume that every node can compute some values $\Delta(h_c) > 0$ and $\gamma(h_c) > 0$ in function of available data in the chain of records ending with the QC hash h_c.

- $\Delta(h_c)$ represents the amount of time available for the first block proposer on top of h_c.
- $\gamma(h_c)$ is the exponent used to increase the time for subsequent proposers.
- As before, we assume that $h_c \approx h'_c$ implies $\Delta(h_c) = \Delta(h'_c)$ and $\gamma(h_c) = \gamma(h'_c)$.

As far as theoretical liveness is concerned, those values could be fixed: $\Delta(h_c) = \Delta_0 > 0$ and $\gamma(h_c) = 2$. However, we expect practical performance to depend on more meaningful values.

7.8. Assigning Durations to Rounds

Assume a latest commit QC hash h_c at round n_c. Let us write $\Delta = \Delta(h_c)$ and $\gamma = \gamma(h_c) > 0$ for the constants mentioned previously.

We define a sequence of *maximum duration* for each $n > n_c + 2$:

$$\texttt{duration}(h_c, n) = \Delta \cdot (n - n_c - 2)^\gamma$$

We make the following observations:

- For the first round after the commit rule, $\texttt{duration}(h_c, n_c + 3) = \Delta$.
- Since as $\Delta > 0$ and $\gamma > 0$, when n grows, $\texttt{duration}(h_c, n)$ keeps increasing and is not bounded.

7.9. Pacemaker State

We may now complete the specifications of the pacemaker state:

```
struct PacemakerState {
    /// Active round
    active_round: Round,
    /// Leader of the active round
    active_leader: Option<Author>,
    /// Time at which we entered the round
    active_round_start: NodeTime,
    /// Nodes known to have switched to the same active round
    active_nodes: HashSet<Author>,
    /// Maximal time allowed between two broadcasts.
    broadcast_interval: Duration,
    /// Maximal duration of the first round after a commit rule.
    delta: Duration,
    /// Exponent to increase round durations.
    gamma: f64,
}
```

7.10. Pacemaker Update Handler

We now describe the implementation of the pacemaker update function seen in Section 7.3 using Rust (Table 3).

The algorithm computes a set of actions to be interpreted by the node as follows:

- After initializing actions with default values, the current active round $\texttt{active\_round}(\alpha)$ is set to $k+1$, where k is the maximum value between:
  - The highest round of a quorum certificate (QC) in $\texttt{record\_store}(\alpha)$, if any;
  - The highest round of a timeout certificate (TC) in $\texttt{record\_store}(\alpha)$, if any; and
  - $0 = \texttt{round}(h_{\texttt{init}})$.
- The current active leader $\texttt{active\_leader}(\alpha)$ is set to $\texttt{leader}(h_c, \texttt{active\_round}(\alpha))$.
- If $\texttt{active\_round}(\alpha)$ was just changed above:
  - Start a timer to track the duration of the round;
  - Reset the list of *active nodes*, defined as the set of nodes which have communicated to us that they shared the same active round as us (this is only useful if we are the leader of this round); and
  - Request that the node notify the new leader $\texttt{active\_leader}(\alpha)$ to be counted as an active node.
- If the round of the `latest_senders` argument is the current active round $\texttt{active\_round}(\alpha)$, then we add the corresponding author to the set of active nodes.
- If $\texttt{active\_leader}(\alpha)$ points to $\texttt{local\_author}(\alpha)$ and the set of active nodes form a quorum, then request that the node fetch a command and propose a block. Also, force a re-evaluation of the main handler so that we vote on our proposal immediately.
- If we have not broadcast in an interval of time I, request a new broadcast.
- If this active round has exceeded its maximal duration and we have not created a timeout yet, then request the creation of a timeout at round $\texttt{active\_round}(\alpha)$ and request a broadcast.
- Finally, reschedule a run of the main handler (hence this function) to a time where we may have to broadcast or create a timeout.

```rust
fn update_pacemaker(
    &mut self,
    local_author: Author,
    record_store: &RecordStore,
    mut latest_broadcast: NodeTime,
    latest_senders: Vec<(Author, Round)>,
    clock: NodeTime,
) -> PacemakerUpdateActions {
    // Initialize actions with default values.
    let mut actions = PacemakerUpdateActions::new();
    // Recompute the active round.
    let active_round = std::cmp::max(
        record_store.highest_quorum_certificate_round(),
        record_store.highest_timeout_certificate_round(),
    ) + 1;
    // If the active round was just updated..
    if active_round > self.active_round {
        // .. store the new value
        self.active_round = active_round;
        // .. start a timer
        self.active_round_start = clock;
        // .. recompute the leader
        self.active_leader = Some(Self::leader(record_store, active_round));
        // .. reset the set of nodes known to have entered this round (useful for leaders).
        self.active_nodes = HashSet::new();
        // .. notify the leader to be counted as an "active node".
        actions.should_notify_leader = self.active_leader;
    }
    // Update the set of "active nodes", i.e. received synchronizations at the same active round.
    for (author, round) in latest_senders {
        if round == active_round {
            self.active_nodes.insert(author);
        }
    }
    // If we are the leader and have seen a quorum of active node..
    if self.active_leader == Some(local_author)
        && record_store.is_quorum(&self.active_nodes)
        && record_store.proposed_block(&*self) == None
    {
        // .. propose a block on top of the highest QC that we know.
        actions.should_propose_block =
            Some(record_store.highest_quorum_certificate_hash().clone());
        // .. force an immediate update to vote on our own proposal.
        actions.should_schedule_update = Some(clock);
    }
    // Enforce sufficiently frequent broadcasts.
    if clock >= latest_broadcast + self.broadcast_interval {
        actions.should_broadcast = true;
        latest_broadcast = clock;
    }
    // If we have not yet, create a timeout after the maximal duration for rounds.
    let deadline = if record_store.has_timeout(local_author, active_round) {
        NodeTime::never()
    } else {
        self.active_round_start + self.duration(record_store, active_round)
    };
    if clock >= deadline {
        actions.should_create_timeout = Some(active_round);
        actions.should_broadcast = true;
    }
    // Make sure this update function is run again soon enough.
    actions.should_schedule_update = Some(std::cmp::min(
        actions.should_schedule_update.unwrap_or(NodeTime::never()),
        std::cmp::min(latest_broadcast + self.broadcast_interval, deadline),
    ));
    actions
}
```

Table 3: Update function of the pacemaker

7.11. Resharing and Cleaning Records

We have sketched the requirements for the data-synchronization protocol between nodes in a previous section (Section 4.4). We are now making more precise the minimal amount of data that must be exchanged during an instance of data synchronization.

Whenever a node α synchronizes with an honest sender α_0, we expect that:

> **(sync-epoch)** The current epoch of α after synchronization is at least as recent as the one of α_0.

Besides, whenever α and α_0 share the same epoch at the end of the synchronization (the usual case), we expect the following properties to hold:

> **(sync-commits)** The highest commit known to α after synchronization is at least as high as the one of α_0.
>
> **(sync-QCs)** The highest QC known to α after synchronization is at least as high as the one of α_0.
>
> **(sync-TCs)** If α_0 knows a TC at a higher round than the highest QC of α, then the highest TC known to α after synchronization is at least as high as the highest TC of α_0.
>
> **(sync-vote)** If α_0 just voted for a block proposed by α, then α receives this vote; no other votes are sent by α_0.
>
> **(sync-timeouts)** If α_0 knows timeouts at its current active round, then α receives these timeouts.
>
> **(sync-block)** If α_0 knows one block proposed by the current leader of its current active round, then α receives this block and learns the chain of previous blocks and QCs.

Regarding the last item (sync-block), note that honest leaders are expected to propose only one block, so we can stop gossiping conflicting proposals.

Regarding the property (sync-epoch), the data-synchronization exchange between the two nodes relies on the past record stores of α_0 (Section 5.6) so that α can follow the chains of commits and the commit rules of all the epochs known to α_0.

A solution for data synchronization is described in Appendix A.3.

Record cleanups. The requirements above condition which data can be cleaned from the record store of receiving nodes after an update.

- We define the *current round* of the record store to be one plus the round of the highest QC or TC in the store.
- In terms of QCs and blocks with QCs, the record store only needs to keep the two chains ending with the highest QC and the last QC of the latest commit rule.
- In terms of blocks without QCs, only one proposal at the current round is needed.
- In terms of votes, if we just proposed a block at the current round, only the votes at the current round are needed; otherwise, only one vote authored by us at the current round is needed.
- In terms of timeouts, only the timeouts at the current active round are needed, together with one set of timeouts that form a TC at the current round minus one, if any such TC exists.

Note that the current round of the record store becomes the active round of the node only after the main handler is called and `update_pacemaker` has started its timer.

Data tracker. The requirements above also translate into the following specifications for the data-tracker abstraction that was used previously in the main handler `update_node` to control resharing (Section 5.7).

```
struct DataTracker {
    /// Latest epoch identifier that was processed.
    epoch_id: EpochId,
    /// Round of the latest commit that was processed.
    highest_committed_round: Round,
    /// Round of the latest highest QC that was processed.
    highest_quorum_certificate_round: Round,
    /// Latest current round of the record store that was processed.
    current_round: Round,
    /// Number of timeouts in the current round.
    num_current_timeouts: usize,
}

impl DataTracker {
    fn update_and_decide_resharing(&mut self, epoch_id: EpochId, record_store: &RecordStore) -> bool {
        let mut should_broadcast = false;
        if epoch_id != self.epoch_id {
            self.epoch_id = epoch_id;
            self.highest_committed_round = Round(0);
            self.highest_quorum_certificate_round = Round(0);
            self.current_round = Round(1);
            self.num_current_timeouts = 0;
            should_broadcast = true;
        }
        let highest_committed_round = record_store.highest_committed_round();
        if highest_committed_round > self.highest_committed_round {
            self.highest_committed_round = highest_committed_round;
            should_broadcast = true;
        }
        let highest_quorum_certificate_round = record_store.highest_quorum_certificate_round();
        if highest_quorum_certificate_round > self.highest_quorum_certificate_round {
            self.highest_quorum_certificate_round = highest_quorum_certificate_round;
            should_broadcast = true;
        }
        let current_round = record_store.current_round();
        if current_round > self.current_round {
            self.current_round = current_round;
            self.num_current_timeouts = 0;
            should_broadcast = true;
        } else {
            let num_current_timeouts = record_store.num_current_timeouts();
            if num_current_timeouts > self.num_current_timeouts {
                self.num_current_timeouts = num_current_timeouts;
                should_broadcast = true;
            }
        }
        should_broadcast
    }
}
```

8. Proof of Liveness

We now consider the liveness of the LibraBFT protocol. We argue that the liveness mechanisms described in Section 7 ensure that commits are being produced in a timely manner whenever the network becomes synchronous.

Without loss of generality, we assume that the current epoch continues indefinitely. We will also rely on the fact that after GST, network and nodes are responsive; hence, we only take into account network propagation delays. Note that we address only the question of chain growth. How committed transactions are picked — aka fairness — is left for future work (see previous note in Section 3.3).

8.1. Active Rounds

In the following, we use *maximal active round* to refer to the maximal active round between honest nodes. Recall that the set of honest nodes is unknown to the participants, yet stays the same during an epoch.

Next, we prove that active rounds are always increasing over time or when nodes synchronize with each other.

Lemma L1: If a node changes its active round from n to n', then $n < n'$.

PROOF: By construction, discarding records (Section 5.7) never decreases the rounds of the highest QC and TC in the record store of a node. Therefore, given the definition of the method `update_pacemaker` (Section 7.10), any change in the active round must result from a higher QC or a higher TC. □

Lemma L2: If a node α has synchronized with a node of active round n in the past, then the active round of α is at least n.

PROOF: This is a consequence of Lemma L1 and the properties of data synchronization (sync-QCs) and (sync-TCs). □

Lemma L3: Assume that the maximal active round amongst honest nodes changes from n to n', then $n' = n + 1$.

PROOF: We have seen with Lemma L1 that the maximal active round cannot decrease. While the maximal active round is n, honest nodes can only vote for blocks or sign timeout objects at round lower or equal than n. The voting power necessary to produce a QC or a TC requires at least one honest node to collaborate; therefore, no QC or TC at a round greater than n can be produced. Given the definition of the method `update_pacemaker` (Section 7.10), this implies $n' \leq n + 1$. □

Note:

- Since active rounds do not decrease by Lemma L1, the definition of the method `proposed_block` used by the main handler (Section 7.3) implies that the first voting constraint (Section 5.4) is always fulfilled the first time that a node wishes to vote on a proposal at a given round.
- The fact that maximal active round increases sequentially is very important for the liveness argument. In particular, given that leaders are predictable until a new commit is produced, after GST, we must not allow malicious nodes and the network to cause a round to be skipped.
- Assuming that nodes agree on the latest commit hash h_c, a similar argument as in the proof of Lemma L3 shows that the maximal active round n will stay the same for at least a time $\texttt{duration}(h_c, n)$ unless a new commit is produced or the leader at round n successfully produces a QC sooner.

8.2. Synchronization of Active Rounds

In the previous section, we discussed necessary conditions for the maximal active round to change. We now aim at sufficient conditions for the *minimal active round* between honest nodes to increase.

Note that by Lemma L1, the minimal active round never decreases.

Let δ_G be the time taken by a full gossip-based broadcast after GST. Assuming that the current minimal active round is n, we say that the system is *synchronized* if the following two conditions hold:

(i) Every honest node with an active round equal to n has initiated a broadcast, including its timeout object at round n, at least once after GST (if any such timeout exists).
(ii) Every honest node with an active round greater than n (if any such node exists) entered its active round less than δ_G time ago.

Note that the last condition can be stated formally as a function on the current states by using the timers in active rounds.

Lemma L4: If an honest node α is first to switch to a new maximal active round $n+1$ at time $t > GST$, then at time $t+\delta_G$, the system is synchronized with a minimal active round at least $n+1$.

PROOF: Given the definition of the method `update_pacemaker` (Section 7.10), α switches round because it learned a QC or a TC at round n. According to the protocol (Section 5.7), this QC or TC was gossiped immediately by α. Since α is first in the round, other honest nodes will fully propagate this QC or TC (or any higher one that might be produced in the meantime); therefore, the assumption on broadcasting after GST applies. By Lemma L2, at time $t+\delta_G$, all honest nodes will have entered an active round greater than n, less than δ_G time ago. Note that no timeout at round greater than n could be created before GST, thus condition (i) holds as well and the system is synchronized. □

Note: Assuming that nodes agree on the latest commit hash h_c and that $\texttt{duration}(h_c, n) \geq \delta_G$, in the condition of Lemma L4, all honest nodes will switch exactly to the round $n+1$ between time t and $t+\delta_G$, unless a new commit is produced or the leader at round $n+1$ successfully produces a QC before $t+\delta_G$.

We can now show that system synchronization persists after GST.

Proposition L5: If the system is synchronized at time $t > GST + \delta_G$, then it stays synchronized at later times $t' > t$.

PROOF: Condition (i) is clearly propagated for old timeouts and true for new ones.

We prove (ii) by contradiction. Let $t' > t$ be the earliest time at which the system is no longer synchronized. Let n_0 be the minimal active round at time t'. By assumption, there exist nodes with an active round $n > n_0$ that switched their active rounds no less than δ_G ago. Let α be the node that switched first to the greatest active round $n_1 > n_0$ amongst those nodes. Since t' is the earliest time of desynchronization, and the minimal active round never decreases, α switched to n_1 exactly at time $t' - \delta_G$. Besides, n_1 was the maximal active round at time $t' - \delta_G$. Since α was the first to switch and $t' - \delta_G > GST$, Lemma L2 applies and shows that the system is synchronized at time t'. □

Lemma L6: If the maximal active round between honest nodes is n at time $t > GST$, then the system is synchronized at time $t + I + \delta_G$, with a minimal active round at least n.

PROOF: Condition (i) follows from the mechanism of regular broadcasts (Section 7.6) with interval I.

The same mechanism also entails that every node α that has the maximal active round n at time t will initiate a broadcast no later than time $t + I$. This broadcast will gossip a QC or TC that justifies the active round n or any later round learned in the meantime. Given Lemma L2 and the assumption on reliable broadcast after GST, every honest node will have an active round at least n by time $t + I + \delta_G$.

Let $n' \geq n$ be the minimal active round at time $t + I + \delta_G$. We prove Condition (ii) by contradiction: assume that there exists a node with an active round $n'' > n'$ and that this node switched to n'' on or before time $t + I$. Since $n'' > n$, any such node must have switched after $t > GST$. By considering

the first node to switch to a new maximal active round, Lemma L4 and Proposition L5 imply that the system is synchronized at time $t + I + \delta_G$. □

Proposition L7: Assume that the highest commit hash h_c is shared between honest nodes and stays the same. If the system is synchronized with a minimal active round at least n at time $t > GST + \delta_G$, then it is synchronized with a minimal active round at least $n+1$ at time $t + 2\delta_G + \texttt{duration}(h_c, n)$.

PROOF: If all honest nodes have an active round greater than n at time t, then the result follows from Lemma L1.

Otherwise, let us assume that the minimal active round at time t is exactly n. If some honest nodes have an active round greater than n at time t, then, by condition (ii), they must have switched less than δ_G time ago. Since $t - \delta_G > GST$, we can conclude by Lemma L4 and Proposition L5 as before.

Otherwise, every honest node has an active round equal to n at time t. By time $t + \texttt{duration}(h_c, n)$, each honest node will have created a timeout object for round n. This timeout object may be created before (or at) time t, or after time t. Given condition (i) of synchronized systems, timeout objects that existed already at time t were already broadcast at least once after GST. Timeout objects created after t are also broadcast immediately by definition of the pacemaker. Given the assumption on gossiping delay after GST, by time $t + \delta_G + \texttt{duration}(h_c, n)$, one honest node will first learn enough old and new timeouts to form a TC at round n, thus switching to the next active round $n + 1$. We then conclude by Lemma L4 and Proposition L5. □

8.3. Optimistic Responsiveness

Following the authors of the original HotStuff [5], we prove an important property for the liveness of the protocol called "Optimistic Responsiveness."

Proposition L8 (Optimistic Responsiveness): Assume that a quorum of nodes (α) communicated their highest 1-chains to a proposer at times (t_α). Let $B \leftarrow C$ be the highest 1-chain amongst all those communicated. If such a node α is honest, we further assume that it has not voted on any proposal since t_α. Then, under BFT assumption, any proposal B' such that $B \leftarrow C \leftarrow B'$ is compatible with the voting rule (locked-round) of any honest node.

PROOF: Let n_0 be the current locked round of an honest node α_0. By definition of the locked round, α_0 once knew a 2-chain $B_0 \leftarrow C_0 \leftarrow B_1 \leftarrow C_1$ such that $\texttt{round}(B_0) = n_0$. Under BFT assumption, there exists an honest node α that voted for B_1 a time t_0 and communicated its highest 1-chain at time t_α. Since α has not voted since t_α, we have $t_0 \leq t_\alpha$. At time t_0, α knew the 1-chain $B_0 \leftarrow C_0$. Since its highest 1-chain at later time t_α is not higher than B and the round of the highest 1-chain in the record store of node never decreases (see record cleanups in Section 7.11), we deduce $n_0 \leq \texttt{round}(B)$. Therefore, α_0 can vote for B' according to the voting rule (locked-round). □

8.4. Main Proof

Let δ_M be the transmission delay for one message after GST.

Theorem L9 (Liveness): Let h_c be the highest commit QC hash known to honest nodes at a time $t_0 > GST$. Let $t_1 = t_0 + I + \delta_G$ and n_1 be the maximal active round at time t_1. Let $n \geq n_1$ be such that $\texttt{leader}(h_c, n)$, $\texttt{leader}(h_c, n+1)$, and $\texttt{leader}(h_c, n+2)$ are honest and such that

$\texttt{duration}(h_c, n) \geq 2\,\delta_M + 2\,\delta_G$. Then, the next commit after h_c is received by every node no later than time:

$$t = t_0 + I + (2\,n - 2\,n_1 + 3)\,\delta_G + \sum_{k=n_1}^{n+2} \texttt{duration}(h_c, k)$$

PROOF: If the highest commit hash h_c known to honest nodes changes before time $t - \delta_G$, then given that $t - \delta_G \geq GST$, the assumption on gossiping after GST implies the next commit after h_c is received by every node no later than time t.

By contradiction, assume that h_c does not change before $t - \delta_G$. Given the regular broadcasts (Section 7.6) and the properties of data synchronization (sync-commits), at time $t_1 = t_0 + I + \delta_G$, all honest nodes have the same highest commit QC hash h_c. From then on and until time $t - \delta_G$, this means every honest node agrees on the leader and the duration of each round. By Lemma L6 and Proposition L5, the system is also synchronized on and after time t_1. By the assumption on gossiping after GST and condition (ii) of synchronization at time t_1, the minimal active round at time $t_2 = t_1 + \delta_G$ is at least n_1.

By Lemma L3, the maximal active rounds on and after t_2 follow the sequence of values n_1, $n_1 + 1$, $n_1 + 2$, etc. Using the minimal active round as a lower bound for the maximal active round, by Lemma L4 and Proposition L7, and given that $\texttt{duration}(h_c, n) > \delta_G$, we deduce that there exists a time t_3 with $t_3 \leq t_2 + 2\,(n - n_1)\,\delta_G + \sum_{k=n_1}^{n-1} \texttt{duration}(h_c, k)$ such that all honest nodes switch to round n between t_3 and $t_3 + \delta_G$. By definition of the pacemaker, these nodes immediately notifies the leader of round n.

Given that $\texttt{duration}(h_c, n) \geq 2\,\delta_M + 2\,\delta_G$, no honest node will timeout and all of them will stay at active round n until the leader at round n has completed its expected tasks:

- Receive the synchronizations from a quorum of nodes (time cost $\leq \delta_M$);
- Pick a valid block compatible with voting rules (see Proposition L8 on optimistic responsiveness);
- Broadcast it (time cost $\leq \delta_G$);
- Gather a quorum of votes (time cost $\leq \delta_M$); and
- Broadcast its QC (time cost $\leq \delta_G$).

Recall that $2\,\delta_M + 2\,\delta_G \leq \texttt{duration}(h_c, n) \leq \texttt{duration}(h_c, n+1) \leq \texttt{duration}(h_c, n+2)$. Therefore, the same reasoning applies to the next leaders at round $n+1$ and $n+2$. This means that all honest nodes will receive three QCs at contiguous rounds n, $n+1$, and $n+2$ – hence a commit, before time $t_4 = t_3 + \delta_G + \sum_{k=n}^{n+2} \texttt{duration}(h_c, k)$.

Using previous equations, we have:

$$t_4 \leq t_0 + I + 2\,\delta_G + 2\,(n - n_1)\,\delta_G + \sum_{k=n_1}^{n-1} \texttt{duration}(h_c, k) + \delta_G + \sum_{k=n}^{n+2} \texttt{duration}(h_c, k) = t$$

□

Note: For subsequent commits after GST, we may assume that the system is already synchronized and that the highest commit known to honest nodes was just broadcast at time $t_0 > GST$. In this case, a similar proof shows that we can spare the term I. Specifically, the next commit will be received no later than $t' = t_0 + (2n' - 2n'_1 + 3)\,\delta_G + \sum_{k=n'_1}^{n'+2} \texttt{duration}(h_c, k)$ when n' is defined as n above, but based on the highest QC round n'_1 at time $t'_1 = t_0 + \delta_G$.

9. Economic Incentives

Finally, we sketch how to economically incentivize LibraBFT nodes for their behaviors in the consensus protocol. Specifically, we show how to reward timely leaders and voters and how to detect violations of voting constraints and conflicting proposals. This covers the essential behaviors of participants. In the future, we intend to study how to cover more behaviors, such as timeouts.

The execution of rewards and punishments is meant to be entirely delegated to the execution module of the Libra Blockchain and programmed using the Move language [38]. Rewards are handled by adding consensus-provided arguments to the execution callbacks. We sketch possible SMR APIs in Appendix A.1.

In the case of punishments, we will rely on a *whistleblower* node to detect a violation, gather cryptographic evidence (see the conditions given below), and submit a *punishment request* through consensus. We leave for future work the exact specifications of the corresponding interactions between mempool, execution, and consensus.

9.1. Leaders and Voters

Assume a proposal B on top of a quorum certificate C_0, that is, $B_0 \leftarrow C_0 \leftarrow B$. Thanks to cryptographic chaining, during the execution of the block B, we may introspect B_0 and C_0 to suggest rewards for the author of B_0 and the authors of the votes included in the quorum certificate C_0.

APIs to communicate lists of authors and voters to the execution are proposed in Appendix A.1. We emphasize that rewards concerning B_0 are computed as part of the speculative execution of some next block B. They become final when B is committed.

Note that we cannot so easily punish unsuccessful leaders $\texttt{leader}(h_c, n)$ for $\texttt{round}(B_0) < n < \texttt{round}(B)$ because there may not be agreement between consensus nodes on h_c, the latest commit preceding B_0.

9.2. Detecting Safety Violations

Looking at the proof of Proposition S4, we notice that the proof of safety relies only on Lemma S2 and Lemma S3.

Interestingly, these two lemmas are merely properties of the tree of records. Therefore, we can translate them into the following conditions to prove that a node α is trying to break safety:

> **(conflicting-votes)** There exist two votes, $B_1 \leftarrow V_1$ and $B_2 \leftarrow V_2$, such that $\texttt{round}(B_1) = \texttt{round}(B_2)$, $\texttt{author}(V_1) = \texttt{author}(V_2) = \alpha$, and either $B_1 \neq B_2$ or $\texttt{state}(V_1) \neq \texttt{state}(V_2)$.
>
> **(locked-round-violation)** There exist a vote following a 2-chain $B_0 \leftarrow C_0 \leftarrow B_1 \leftarrow C_1 \leftarrow B_2 \leftarrow V_2$ and a vote $B \leftarrow V$, such that $\texttt{author}(V_2) = \texttt{author}(V) = \alpha$, $\texttt{round}(B) > \texttt{round}(B_2)$, and $\texttt{previous\_round}(B) < \texttt{round}(B_0)$.

Proposition E1 (Safe detection): A node that respects the voting rules (increasing-round) and (locked-round) never triggers the conditions (conflicting-votes) and (locked-round-violation).

Proof: This was proved as part of the proofs of Lemma S2 and Lemma S3, respectively. □

Proposition E2 (Complete detection): If no more than f nodes ever triggered the conditions (conflicting-votes) and (locked-round-violation), then safety holds.

PROOF: As mentioned above, the proofs of Proposition S4, or safety, rely only on Lemma S2 and Lemma S3. We prove these lemmas under the new assumption by considering a non-violating node (instead of an honest node) at the intersection of the two QCs mentioned at the beginning of the original proofs. □

9.3. Detecting Conflicting Proposals

According to the protocol, the leader of a round should make only one proposal. Making several proposals does not endanger safety, but it makes other nodes consume more resources than needed (e.g., CPU, network). This undesirable behavior is easy to detect:

> **(conflicting-proposals)** There exist two proposals B_1 and B_2 such that $\texttt{round}(B_1) = \texttt{round}(B_2)$, $B_1 \neq B_2$, and $\texttt{author}(B_1) = \texttt{author}(B_2) = \alpha$.

10. Conclusion

We have presented LibraBFT, a state machine replication system based on the HotStuff protocol [5] and designed for the Libra Blockchain [2]. LibraBFT provides safety and liveness in a Byzantine setting when up to one-third of voting rights are held by malicious actors, assuming that the network is partially synchronous. In this report, we have presented detailed proofs of safety and liveness and covered many important practical considerations, such as networking and data structures. We have shown that LibraBFT is compatible with proof of stake and can generate incentives for a variety of behaviors, such as proposing blocks and voting. Thanks to the simplicity of the safety argument in LibraBFT, we also provided criteria to detect malicious attempts to break safety. These criteria will be instrumental for the progressive migration of the Libra infrastructure to a permissionless model.

Future work. This report constitutes an initial proposal for LibraBFT and is meant to be updated in the future. In the next version, we intend to share the code for our reference implementation in a simulated environment and provide experimental results, both using this simulation and using the production implementation currently developed by Calibra engineers.

In the future, we would like to improve our theoretical analysis in several ways. We plan to make our networking assumptions more precise, with additional studies on message sizes and probabilistic gossiping. Regarding the integration of LibraBFT with the Libra Blockchain, we would like to cover fairness and discuss how light clients can authenticate the set of validators for each epoch. Economic incentives should reward additional positive behaviors, such as creating timeouts, and specifications should provide an external protocol for auditors to report violations of safety rules.

On a practical level, we have not yet analyzed resource consumption (memory, CPU, etc.) in the presence of malicious participants. Heuristics for leader selection, a precise description of the VRF solution, and possibly adaptive policies will likely be required to increase the robustness of the system in case of malicious leaders or targeted attacks on leaders.

In the long term, we hope that our efforts on precise specifications and detailed proofs will pave the way for mechanized proofs of safety and liveness of LibraBFT.

Acknowledgments

We would like to thank the following people for helpful discussions and feedback on this paper: Tarun Chitra, Ittay Eyal, Klaus Kursawe, John Mitchell, and Jared Saia.

A. Programming Interfaces

Note: This section will evolve in the future as we integrate engineering optimizations and make progress in the software implementation of LibraBFT.

A.1. State Machine Replication

We now present possible programming interfaces for state machine replication (Table 4).

We assume two abstract data types:

- Values of type `State` are authenticators that refer to a concrete execution state in the Libra Blockchain.
- `Command` values are meant to be executed on top of a `State` value.

At the beginning of the first epoch, we assume that the SMR module of every node is initialized with the same initial value of type `State`. As mentioned above (Section 3), in practice, `State` values contain a hash value that points to a persistent local storage outside the SMR module.

The SMR module communicates with the other modules of a Libra validator through a number of APIs (i.e., Rust traits):

- `CommandFetcher` lets the SMR module fetch user commands from the mempool.
- `StateComputer` produces a new state hash from the hash of a base state, a command to execute, and additional contextual data, including a proposed system time and signals for economic incentives (Section 9).
- `StateFinalizer` lets the SMR module eventually declare whether each state hash was successfully committed or not. In the case of a commit, we pass the quorum certificate that contains the corresponding `commitment` value, as discussed in Section 4.1.
- `EpochReader` lets the SMR module retrieve a possibly updated epoch identifier from a state, as well as the current voting rights.

A.2. Record Store

The implementation of the record store is assumed to provide the APIs outlined in Table 5.

In the simulator used as a reference for this report, we implement the `RecordStore` APIs using the in-memory data structures described in Table 6. Note that the data structures described here do not cover constant-time cleanups, persistent storage, and resistance to potential crashes while the record store is being updated.

A.3. Data-Synchronization Messages

The messages of the data-synchronization protocol (Section 4.6) used in our current simulator are described in Table 7. For simplicity, we have assumed that data are transmitted over authenticated channels and omitted message signatures.

```rust
trait CommandFetcher {
    /// How to fetch valid commands to submit to the consensus protocol.
    fn fetch(&mut self) -> Command;
}

trait StateComputer {
    /// How to execute a command and obtain the next state.
    /// If execution fails, the value `None` is returned, meaning that the
    /// command should be rejected.
    fn compute(
        &mut self,
        // The state before executing the command.
        base_state: &State,
        // Command to execute.
        command: Command,
        // Time associated to this execution step, in agreement with
        // other consensus nodes.
        time: NodeTime,
        // Suggest to reward the author of the previous block, if any.
        previous_author: Option<Author>,
        // Suggest to reward the voters of the previous block, if any.
        previous_voters: Vec<Author>,
    ) -> Option<State>;
}

/// How to communicate that a state was committed or discarded.
trait StateFinalizer {
    /// Report that a state was committed.
    fn commit(&mut self, state: &State, certificate: Option<&QuorumCertificate>);

    /// Report that a state was discarded.
    fn discard(&mut self, state: &State);
}

/// Hold voting rights for a give epoch.
struct EpochConfiguration {
    voting_rights: BTreeMap<Author, usize>,
    total_votes: usize,
}

/// How to communicate that a state was committed or discarded.
trait EpochReader {
    /// Read the id of the epoch in a state.
    fn read_epoch_id(&self, state: &State) -> EpochId;

    /// Return the configuration (i.e. voting rights) for the current epoch.
    fn configuration(&self) -> EpochConfiguration;
}

trait SMRContext: CommandFetcher + StateComputer + StateFinalizer + EpochReader {}
```

Table 4: Programming interfaces for State Machine Replication

```rust
trait RecordStore {
    /// Return the hash of a QC at the highest round, or the initial hash.
    fn highest_quorum_certificate_hash(&self) -> QuorumCertificateHash;
    /// Query the round of the highest QC.
    fn highest_quorum_certificate_round(&self) -> Round;
    /// Query the round of the highest TC.
    fn highest_timeout_certificate_round(&self) -> Round;
    /// Query the round of highest commit.
    fn highest_committed_round(&self) -> Round;
    /// Query the round of the highest 2-chain.
    fn highest_2chain_head_round(&self) -> Round;

    /// Current round as seen by the record store.
    fn current_round(&self) -> Round;
    /// Number of timeouts objects known at the current round.
    fn num_current_timeouts(&self) -> usize;

    /// Iterate on a chain of QCs starting after the QC at round `after_round` and ending with the QC at round
    /// `until_round`
    fn chain_between_quorum_certificates<'a>(
        &'a self,
        after_round: Round,
        until_round: Round,
    ) -> ForwardQuorumCertificateIterator<'a>;
    /// Find a QC whose `commitment` field is the state of the input QC.
    fn commit_certificate(&self, qc: &QuorumCertificate) -> Option<&QuorumCertificate>;

    /// Access the block proposed by the leader chosen by the Pacemaker (if any).
    fn proposed_block(&self, pacemaker: &Pacemaker) -> Option<(BlockHash, Round, Author)>;
    /// Check if a timeout already exists.
    fn has_timeout(&self, author: Author, round: Round) -> bool;

    /// Create a timeout.
    fn create_timeout(&mut self, author: Author, round: Round, smr_context: &mut SMRContext);
    /// Fetch a command from mempool and propose a block.
    fn propose_block(
        &mut self,
        local_author: Author,
        previous_qc_hash: QuorumCertificateHash,
        clock: NodeTime,
        smr_context: &mut SMRContext,
    );
    /// Execute the command contained in a block and vote for the resulting state.
    /// Return false if the execution failed.
    fn create_vote(
        &mut self,
        local_author: Author,
        block_hash: BlockHash,
        smr_context: &mut SMRContext,
    ) -> bool;
    /// Try to create a QC for the last block that we have proposed.
    fn check_for_new_quorum_certificate(
        &mut self,
        local_author: Author,
        smr_context: &mut SMRContext,
    ) -> bool;

    /// Compute the previous round of a block.
    fn previous_round(&self, block_hash: BlockHash) -> Round;
    /// Determine if a set of nodes form a quorum.
    fn is_quorum(&self, authors: &HashSet<Author>) -> bool;
    /// Pick an author based on a seed, with chances proportional to voting rights.
    fn pick_author(&self, seed: u64) -> Author;

    /// APIs supporting data synchronization.
    fn highest_commit_certificate(&self) -> Option<&QuorumCertificate>;
    fn highest_quorum_certificate(&self) -> Option<&QuorumCertificate>;
    fn timeouts(&self) -> Vec<Timeout>;
    fn current_vote(&self, local_author: Author) -> Option<&Vote>;
    fn block(&self, block_hash: BlockHash) -> Option<&Block>;
    fn known_quorum_certificate_rounds(&self) -> BTreeSet<Round>;
    fn verify_quorum_certificate_without_previous_block(&self, qc: &QuorumCertificate) -> Result<()>;
    fn unknown_records(&self, known_qc_rounds: BTreeSet<Round>) -> Vec<Record>;
    fn insert_network_record(&mut self, record: Record, smr_context: &mut SMRContext);
}
```

Table 5: Programming interfaces for the record store

```
struct RecordStoreState {
    /// Epoch initialization.
    epoch_id: EpochId,
    configuration: EpochConfiguration,
    initial_hash: QuorumCertificateHash,
    initial_state: State,
    /// Storage of verified blocks and QCs.
    blocks: HashMap<BlockHash, Block>,
    quorum_certificates: HashMap<QuorumCertificateHash, QuorumCertificate>,
    round_to_qc_hash: HashMap<Round, QuorumCertificateHash>,
    current_proposed_block: Option<BlockHash>,
    /// Computed round values.
    highest_quorum_certificate_round: Round,
    highest_timeout_certificate_round: Round,
    current_round: Round,
    highest_2chain_round: Round,
    highest_committed_round: Round,
    /// Storage of verified timeouts at the highest TC round.
    highest_timeout_certificate: Option<Vec<Timeout>>,
    /// Storage of verified votes and timeouts at the current round.
    current_timeouts: HashMap<Author, Timeout>,
    current_votes: HashMap<Author, Vote>,
    /// Computed weight values.
    current_timeouts_weight: usize,
    current_election: ElectionState,
}

/// Counting votes for a proposed block and its execution state.
enum ElectionState {
    Ongoing { ballot: HashMap<(BlockHash, State), usize> },
    Won { block_hash: BlockHash, state: State },
    Closed,
}
```

Table 6: In-memory data structures for the record store

```
struct DataSyncNotification {
    /// Current epoch identifier.
    current_epoch: EpochId,
    /// Tail QC of the highest commit rule.
    highest_commit_certificate: Option<QuorumCertificate>,
    /// Highest QC.
    highest_quorum_certificate: Option<QuorumCertificate>,
    /// Timeouts in the highest TC, then at the current round, if any.
    timeouts: Vec<Timeout>,
    /// Sender's vote at the current round, if any (meant for the proposer).
    current_vote: Option<Vote>,
    /// Known proposed block at the current round, if any.
    proposed_block: Option<Block>,
    /// Active round of the sender's pacemaker.
    active_round: Round,
}

struct DataSyncRequest {
    /// Current epoch identifier.
    current_epoch: EpochId,
    /// Selection of rounds for which the receiver already knows a QC.
    known_quorum_certificates: BTreeSet<Round>,
}

struct DataSyncResponse {
    /// Current epoch identifier.
    current_epoch: EpochId,
    /// Records for the receiver to insert, for each epoch, in the given order.
    records: Vec<(EpochId, Vec<Record>)>,
    /// Active round of the sender's pacemaker.
    active_round: Round,
}
```

Table 7: Data-synchronization messages

References

[1] The Libra Association, "An Introduction to Libra." https://libra.org/en-us/whitepaper.

[2] Z. Amsden *et al.*, "The Libra Blockchain." https://developers.libra.org/docs/the-libra-blockchain-paper.

[3] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS'82)*, vol. 4, no. 3, pp. 382–401, 1982.

[4] S. Bano *et al.*, "Moving toward permissionless consensus." https://libra.org/permissionless-blockchain.

[5] M. Yin, D. Malkhi, M. K. Reiterand, G. G. Gueta, and I. Abraham, "HotStuff: BFT consensus in the lens of blockchain," 2019. http://arxiv.org/abs/1803.05069v4

[6] M. Yin, D. Malkhi, M. K. Reiterand, G. G. Gueta, and I. Abraham, "HotStuff: BFT consensus with linearity and responsiveness," in *38th ACM symposium on Principles of Distributed Computing (PODC'19)*, 2019.

[7] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. http://bitcoin.org/bitcoin.pdf

[8] digiconomist.net, https://digiconomist.net/bitcoin-energy-consumption

[9] arewedecentralizedyet.com, https://arewedecentralizedyet.com/

[10] C. Cachin and M. Vukolić, "Blockchain consensus protocols in the wild," 2017. https://arxiv.org/abs/1707.01873

[11] S. Bano *et al.*, "Consensus in the age of blockchains," 2017. https://arxiv.org/abs/1711.03936

[12] I. Abraham, D. Malkhi, and others, "The blockchain consensus layer and BFT," *Bulletin of EATCS*, vol. 3, no. 123, 2017.

[13] M. Ben-Or, "Another advantage of free choice: Completely asynchronous agreement protocols," in *2nd ACM symposium on Principles of Distributed Computing (PODC'83)*, 1983, pp. 27–30.

[14] P. Feldman and S. Micali, "Optimal algorithms for byzantine agreement," in *20th annual ACM symposium on Theory of Computing*, 1988, pp. 148–161.

[15] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of BFT protocols," in *23rd ACM SIGSAC conference on Computer and Communications Security (CCS'16)*, 2016, pp. 31–42.

[16] R. Canetti and T. Rabin, "Fast asynchronous byzantine agreement with optimal resilience," in *25th annual ACM symposium on Theory of Computing (STOC'93)*, 1993, pp. 42–51.

[17] I. Abraham, D. Malkhi, and A. Spiegelman, "Validated asynchronous byzantine agreement with optimal resilience and asymptotically optimal time and word communication." 2018. https://arxiv.org/abs/1811.01332

[18] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

[19] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *3rd symposium on Operating Systems Design and Implementation (OSDI'99)*, 1999, vol. 99, pp. 173–186.

[20] A. Bessani, J. Sousa, and E. E. P. Alchieri, "State machine replication for the masses with BFT-SMART," in *44th annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'14)*, 2014, pp. 355–362.

[21] E. Androulaki *et al.*, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *13th EuroSys conference (EuroSys'18)*, 2018, p. 30.

[22] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," in *21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*, 2007, pp. 45–58.

[23] A. Clement *et al.*, "Upright cluster services," in *22nd ACM Symposium on Operating Systems Principles (SOSP'09)*, 2009, pp. 277–290.

[24] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming.* 2011.

[25] M. K. Reiter, "The rampart toolkit for building high-integrity services," in *Theory and practice in distributed systems*, 1995, pp. 99–110.

[26] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing bitcoin security and performance with strong consistency via collective signing," in *25th USENIX security symposium (USENIX security '16)*, 2016, pp. 279–296.

[27] G. G. Gueta *et al.*, "SBFT: A scalable decentralized trust infrastructure for blockchains," 2018. https://arxiv.org/abs/1804.01626

[28] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the Weil pairing," in *Advances in Cryptology (ASIACRYPT 2001)*, 2001, pp. 514–532.

[29] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," 2018. http://arxiv.org/abs/1807.04938v2

[30] V. Buterin and V. Griffith, "Casper, the friendly finality gadget," 2017. https://arxiv.org/abs/1710.09437

[31] T. H. Chan, R. Pass, and E. Shi, "PaLa: A simple partially synchronous blockchain." 2018. https://eprint.iacr.org/2018/981

[32] Y. Sompolinsky and A. Zohar, "Secure high-rate transaction processing in bitcoin," in *19th international conference on financial cryptography and data security (FC'15)*, 2015, pp. 507–527.

[33] C. Li, P. Li, W. Xu, F. Long, and A. C.-c. Yao, "Scaling Nakamoto consensus to thousands of transactions per second," 2018. https://arxiv.org/abs/1805.03870

[34] G. Danezis and D. Hrycyszyn, "Blockmania: From block DAGs to consensus," 2018. http://arxiv.org/abs/1809.01620

[35] "The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance," 2016.

[36] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, pp. 299–319, 1990.

[37] D. Malkhi and M. Reiter, "Byzantine quorum systems," in *29th annual ACM symposium on Theory of Computing (STOC'97)*, 1997.

[38] S. Blackshear *et al.*, "Move: A language with programmable resources." https://developers.libra.org/docs/move-paper.

[39] S. Micali, M. Rabin, and S. Vadhan, "Verifiable random functions," in *40th annual Symposium on Foundations of Computer Science (FOCS'99)*, 1999, pp. 120–130.