State Machine Replication in the Libra Blockchain

Shehar Bano, 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.

Revised September 25, 2019

Initially, the participating validators will be permitted into the consensus network by an association consisting of a geographically distributed and diverse set of 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. All the code fragments in Rust are directly extracted from our reference implementation in a discrete-event simulated environment [36].

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 [37] 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 $f = \lfloor \frac{N-1}{3} \rfloor$.

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 [38]:

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 $m_1 = m_2$.

When hashing and signing data structures (e.g. in Section 4.1), we assume that all the data fields except signatures, preceded by a type tag, are first turned into a hash value then signed.

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, as long as they do not lose local state data and become responsive again by the end of the asynchronous period. During periods of synchrony, we assume that there exists an upper bound δ to the transmission delay taken by any message between honest nodes, and that honest processes are responsive and do not crash. We stress that the adversary controls malicious nodes and the scheduling of networking messages even during periods of synchrony, subject to the maximal delay δ.

The parameters of this model — such as the value of δ 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 as follows:

> **(eventually-synchronous-network)** After GST, the network delivers all messages between honest nodes within some (unknown) time delay $\delta > 0$.
>
> **(eventually-no-crash)** After GST, honest nodes are responsive and do not crash.

We remark that the GST model does not take into account CPU time associated with message processing. Since message sizes in LibraBFT are not bounded, a fixed δ is arguably over-simplified. In future work, we may enforce strict bounds on message sizes or take into consideration message sizes in modeling transmission delays, e.g., assume a delay that consists of a fixed latency component plus a size-dependent component.

2.6. Leaders, Rounds, Blocks, Votes

LibraBFT belongs to the family of *leader-based* consensus protocols. In leader-based protocols, nodes make progress in rounds, each round having a designated node called a *leader*. Leaders are responsible for proposing new commands and obtaining signatures, called *votes*, from other nodes 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 of *blocks* using cryptographic hashes.

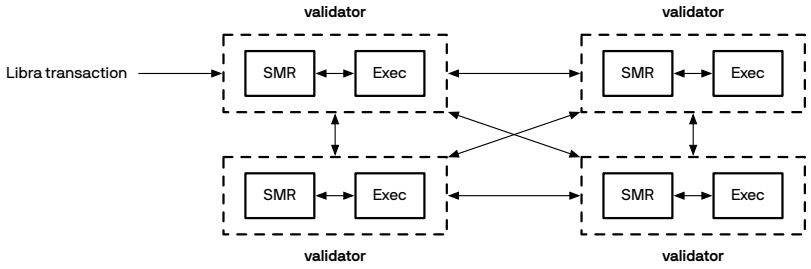

Figure 1: Integration of the SMR module into the Libra Blockchain

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 [39], 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) [40] 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

We have outlined the core concepts of LibraBFT in Section 2.6. Before proceeding to a more precise description of the protocol followed by LibraBFT nodes (Section 5), we define chained data structures, called *records*, meant to be exchanged between nodes and stored as part of their states. We also present the communication framework used to synchronize sets of records over the network.

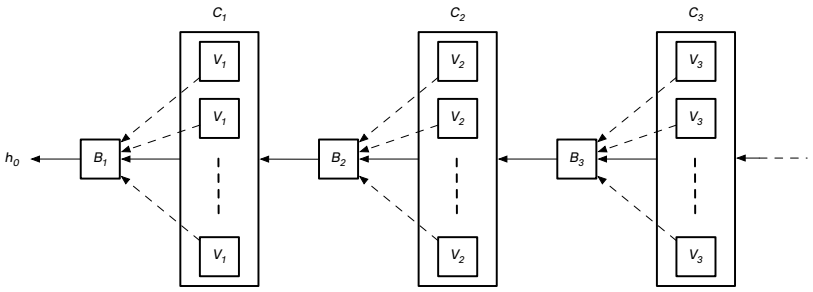

Figure 2: A chain of records in LibraBFT

4.1. Records

The core state of a LibraBFT node consists of a collection of *records* that are stored locally and continuously exchanged over the network. We define four kinds of records:

- *blocks*, meant for the leader of a given round to propose 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.

All records are signed by their *authors*. Importantly, blocks are *chained* (Figure 2): they must include the hash of the QC of a *previous block* at a lower round — except for the initial block of an epoch, which uses a fixed hash value $h_{\texttt{init}}$ as previous QC hash. 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 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.

Data structures. The records of LibraBFT are specified using Rust syntax in Table 1, assuming the following primitive data types:

- `State` (an abstract SMR execution state, e.g., a reference to local storage)
- `Command` (an abstract SMR command, e.g., a sequence of transactions)
- `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).

Record Store. 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 invariants defined in the next section (Section 4.2). However, we generally use *records* for *verified records* when the context is clear.

For each node, the data structure holding the verified records of a single 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.

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. We sketch the possible interfaces of the record store object in Appendix A.2. We make

```
/// 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.
    committed_state: 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.
    committed_state: 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,
    /// Round of the highest block with a quorum certificate.
    highest_certified_block_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

precise 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.

Commitments for Libra clients. When a node votes on a block B, it must set the `committed_state` field of the `Vote` record to a non-empty value whenever forming a QC on B would result in a new commit (according to the commit rule in Section 5.3 below). Say that an ancestor B' of B will be newly committed. Then, `committed_state` is set to the execution state resulting from executing the chain up to B' (included). Note that this execution state is already included in the QC of B'. Embedding it again in the `committed_state` field of the QC of B provides clients with a *commit certificate* for B' — that is, a short cryptographic proof the execution state after B' was committed.

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`.

A consensus node must sequentially verify all the records that it receives before inserting them to its record store:

- 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.
- Timeout objects must contain a highest certified block round that is not greater than the highest QC round verified so far.
- The `committed_state` value in a vote or in quorum certificate should be consistent with the commit rule (Section 5.3).
- 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).

4.3. Communication Framework

In LibraBFT, the communication framework aims at sharing records with a subset of nodes. Specifically, the framework API consists of three primitive actions:

(i) *send*, where the node shares the records it has with a fixed number of peers;
(ii) *broadcast*, where the node sends (*pushes*) updates to every other node;
(iii) *query-all* where a node requests (*pulls*) updates from every other node.

We note that the broadcast action is only guaranteed to reach all honest nodes (aka are *reliable* [24]) after GST and if the sender is honest. Before GST or if the sender is dishonest, data may not be received by every node within the expected delay δ. To address this difficulty, LibraBFT nodes trigger a query-all action periodically when they receive no new commits for a long time, or when their current round stays the same after the timeout period is expired (see Sections 5.6, 7.10

and 7.11). The diffusion of timeouts and the two periodic mechanisms are the only broadcast and query-all actions triggered by non-leaders. By limiting all-to-all communication to these delayed mechanisms, we aim to ensure that in the favorable case where leaders are honest and the network is fast enough, LibraBFT only requires a linear number of point-to-point communication steps to produce a commit.

4.4. Data Synchronization

To support the restarting of crashed nodes and the addition of new nodes, we wish to synchronize data in a flexible way, where senders initiate communication but receivers can skip data that they already have. In LibraBFT, this is addressed by introducing an exchange protocol called *data synchronization*.

In the case of the send and broadcast actions ((i) and (ii) above) data synchronization is done in multiple steps:

- The sender makes the new data available (i.e., passively publish it) as part of its *data-synchronization service*.
- It then sends a *notification* message to each receiver according to the nature of the communication: a single receiver for point-to-point communications, or all nodes in case of a broadcast action.
- Receivers optionally connect back to the sender with a *request* message and retrieve data contained in a *response* message.

Query-all actions ((iii) above) consist of the third step only: the receiving node sends a request message to every other node.

When an exchange is completed, receivers should validate received data immediately (Section 4.2) then make all valid and *relevant* data available for future exchanges. The nature of the relevant data that must be re-shared in this way is made precise in Section 7.12.

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 notification, request, and response messages sketched in Section 4.4 to their intended recipient in an authenticated channel. We create three corresponding handlers to be called when a message is received and returning a possible answer. The additional methods `create_notification` and `create_request` are used when the main handler of node (Section 5.6) requests that the runtime environment initiate one of the network actions discussed in Section 4.3.

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 with a receiver.
    fn create_notification(&self) -> DataSyncNotification;
    /// Query role: what to send to initiate a query exchange and obtain data from a sender.
    fn create_request(&self) -> DataSyncRequest;
    /// Sender role: handle a request from a receiver.
    fn handle_request(&self, request: DataSyncRequest) -> DataSyncResponse;
    /// Receiver role: accept or refuse a notification.
    fn handle_notification(
        &mut self,
        notification: DataSyncNotification,
        context: &mut SMRContext,
    ) -> Option<DataSyncRequest>;
    /// Receiver role: receive data.
    fn handle_response(&mut self, response: DataSyncResponse, context: &mut SMRContext, clock: NodeTime);
}
```

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 `DataSyncNotification`, `DataSyncRequest`, and `DataSyncResponse` 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 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 $R_0 \leftarrow R_1 \ldots \leftarrow R_n$, $n \geq 0$.

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

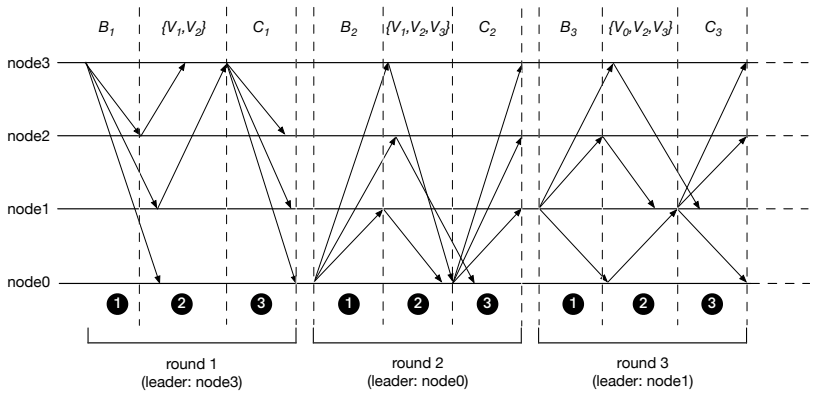

Figure 3: Overview of the LibraBFT protocol (simplified, excluding round synchronization)

blocks B_i and quorum certificates C_i. Formally, such as a chain is denoted: $h_{\texttt{init}} \leftarrow B_1 \leftarrow C_1 \ldots \leftarrow B_n[\leftarrow C_n]$.

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: $h_{\texttt{init}} \leftarrow B_1 \leftarrow C_1 \ldots \leftarrow B_{n+1} \leftarrow C_{n+1}$. 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 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 LibraBFT 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 on commitments (Section 4.1), a valid quorum certificate in the position of C_2 acts as a *commit certificate*: it must include a non-empty field value `committed_state` to authenticate that the execution state $\texttt{state}(C_0)$ was committed in the current epoch. Note that if `committed_state` 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

We 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, $\texttt{previous\_round}(B) = 0$.

Further, we define the *second-previous round* of a block B as $\texttt{second\_previous\_round}(B) = \texttt{round}(B'')$ if there exist B'', C'', B' and C' such that $B'' \leftarrow C'' \leftarrow B' \leftarrow C' \leftarrow B$, and $\texttt{second\_previous\_round}(B) = 0$ otherwise.

The *locked round* of a node α, denoted $\texttt{locked\_round}(\alpha)$, is the highest second-previous round $\texttt{second\_previous\_round}(B)$ over all the blocks B that α ever voted for, if any, and zero otherwise. In other words, at the beginning of an epoch, a node α initializes the value $\texttt{locked\_round}(\alpha)$ to 0 and update it to $\max(\texttt{locked\_round}(\alpha), \texttt{second\_previous\_round}(B))$ whenever it votes for a block B.

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) — denoted $\texttt{record\_store}(\alpha)$ — 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 round synchronization (aka leader election). We group them into a special object called a *pacemaker* and describe them in Section 7.3.

Additional state variables needed by a LibraBFT node include:

- The current epoch identifier $\texttt{epoch\_id}(\alpha)$, used to detect the end of the current epoch;
- The identifier of α as an author of records, denoted $\texttt{local\_author}(\alpha)$;
- The round of the latest voted block $\texttt{latest\_voted\_round}(\alpha)$, (initial value: 0);
- The locked round $\texttt{locked\_round}(\alpha)$, (initial value: 0); and
- The system time of the latest query-all operation $\texttt{latest\_query\_all\_time}(\alpha)$, (initial value: the starting time of the epoch).

The variable $\texttt{latest\_query\_all\_time}(\alpha)$ above is related to liveness. It is used to trigger query-all network actions when not enough progress is made in terms of learning new commits and new rounds, respectively (see Section 4.3 and Section 7).

The state of a node also includes an object, called *commit tracker*, responsible for tracking commits that the main handler has already processed and deciding if a query-all action is needed to recover missing commits. We give more details about the commit 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 particular nodes;
- It may ask to broadcast data notifications; and
- It may ask to pull data from every node (aka query-all action).

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 the latest query-all operation.
    latest_query_all_time: NodeTime,
    /// Track data to which the main handler has already reacted.
    tracker: CommitTracker,
    /// 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, 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 object (`context`) to support SMR operations such as command execution.

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.
    next_scheduled_update: NodeTime,
    /// Whether we need to send a notification to a subset of nodes.
    should_send: Vec<Author>,
    /// Whether we need to send a notification to all other nodes.
    should_broadcast: bool,
    /// Whether we need to request data from all other nodes.
    should_query_all: 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 main handler `update_node` of a LibraBFT node 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 has proposed a block and if a quorum of votes for the same state was just received, create a quorum certificate, trigger a broadcast, and schedule a new run of `update_node` immediately.
- Process newly found commits by calling the method `process_commits` (see below).
- Run the commit tracker module to update its state and decide if the node should perform a query-all action.
- Update the time of latest query-all operation.

The main handler 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 `CommitTracker` object provides the latest commit processed so far, as well as a method `update_tracker` to update the latest epoch and commit values and return necessary action items, notably the query-all action (Section 7.11).

Finally, the algorithm `process_commits` (Table 3) aims to

- deliver commits to the state machine replication context,
- check for a commit that would terminate the current epoch, and
- start a new epoch with the right parameter if needed.

This algorithm relies on the record store of a node for computing the round of the highest committed block `highest_committed_round` in function of newly received data. This value is compared to the corresponding field of the commit tracker to decide if new commits must be delivered. The method `committed_states_after` takes a round m and the record store as inputs. Assuming that the record store contains a chain $B_1 \leftarrow C_1 \leftarrow \ldots \leftarrow B_k \leftarrow C_k$ such that $\texttt{round}(B_1) > m$ is minimal and B_k is the highest committed block, the method `committed_states_after` returns an iterator on the rounds and the states $(\texttt{round}(B_1), \texttt{state}(C_1)), \ldots, (\texttt{round}(B_k), \texttt{state}(C_k))$, in this order. The *highest commit certificate* `highest_commit_certificate` of the record store is defined as the last QC of the commit rule of the highest committed block. (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 pacemaker_actions = self.pacemaker.update_pacemaker(
            self.local_author,
            self.epoch_id,
            &self.record_store,
            self.latest_query_all_time,
            clock,
        );
        let mut actions = self.process_pacemaker_actions(pacemaker_actions, clock, smr_context);
        // 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 the latest voted round.
                self.latest_voted_round = block_round;
                // Update the locked round.
                self.locked_round =
                    max(self.locked_round, self.record_store.second_previous_round(block_hash));
                // 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 to notify and send our vote to the author of the block.
                    actions.should_send = vec![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) {
            // Broadcast the QC to finish our work as a leader.
            actions.should_broadcast = true;
            // Schedule a new run now to process the new QC.
            actions.next_scheduled_update = clock;
        }
        // Check for new commits and verify if we should start a new epoch.
        self.process_commits(smr_context);
        // Update the commit tracker and ask that we query all nodes if needed.
        let tracker_actions =
            self.tracker.update_tracker(self.latest_query_all_time, clock, self.epoch_id, &self.record_store);
        actions.should_query_all = actions.should_query_all || tracker_actions.should_query_all;
        actions.next_scheduled_update =
            min(actions.next_scheduled_update, tracker_actions.next_scheduled_update);
        // Update the time of the latest query-all action.
        if actions.should_query_all {
            self.latest_query_all_time = clock;
        }
        // Return desired actions to main handler.
        actions
    }
}
```

Table 2: Main handler of a LibraBFT node

```
impl NodeState {
    fn process_commits(&mut self, smr_context: &mut SMRContext) {
        // For all commits that have not been processed yet, according to the commit tracker..
        for (round, state) in self.record_store.committed_states_after(self.tracker.highest_committed_round) {
            // .. deliver the committed state to the SMR layer, together with a commit certificate,
            // if any.
            if round == self.record_store.highest_committed_round() {
                smr_context.commit(&state, self.record_store.highest_commit_certificate())
            } else {
                smr_context.commit(&state, None);
            };
            // .. check if the current epoch just ended. If it did..
            let new_epoch_id = smr_context.read_epoch_id(&state);
            if new_epoch_id > self.epoch_id {
                // .. create a new record store and switch to the new epoch.
                let new_record_store = RecordStoreState::new(
                    new_epoch_id.initial_hash(),
                    state.clone(),
                    new_epoch_id,
                    smr_context.configuration(&state),
                );
                let old_record_store = std::mem::replace(&mut self.record_store, new_record_store);
                self.past_record_stores.insert(self.epoch_id, old_record_store);
                self.epoch_id = new_epoch_id;
                // .. initialize voting constraints.
                self.latest_voted_round = Round(0);
                self.locked_round = Round(0);
                // .. stop delivering commits after an epoch change.
                break;
            }
        }
    }
}
```

Table 3: Processing new commits

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.

We have stated safety rules for a single epoch. This formulation is sufficient to reason about safety given that nodes only participate (e.g. vote) in the newest epoch and never roll back to an old epoch. It is also understood that the initialization parameters of a new epoch must be uniquely determined by the committed execution state that has created the epoch.

To ensure liveness, nodes must keep the record stores of the previous epochs available: 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. Technically, this means that the method `process_commits` may be called, an old epoch may be stopped, and a new one started during data synchronization (see also Section 3.1 and Section 7.12).

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 leaders 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. Under BFT assumption, for every certified block $B \leftarrow C$ such that $\texttt{round}(B) > n_2$, we have that $\texttt{previous\_round}(B) \geq n_0$.

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 updated its locked round to be at least $\texttt{second\_previous\_round}(B_2) = \texttt{round}(B_0) = n_0$. Since the locked round never decreases, at the later time of voting for B, the voting rule (locked-round) of α 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. Under BFT assumption, for every certified block $B \leftarrow C$ such that $\texttt{round}(B) \geq n_0$, we have that $B_0 \leftarrow^* B$.

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 round synchronization (including leader election and the decision when to enter a new round) to a special module called a *pacemaker*. We now describe the pacemaker of LibraBFT in detail, as well as the policies for re-sharing records, cleaning the record store, and tracking commits to decide additional query-all operations. 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, each created by a distinct author, such that the set of timeout authors form a quorum (Section 2.3). The round of a timeout certificate is the common round value n.

Intuitively, TCs are to timeout objects what QCs are to votes. However, TCs are neither materialized as records nor chained in the record store.

7.2. Overview of the Pacemaker

When a node enters a new round n, it computes the leader and the maximal duration of that round using two predefined functions $\texttt{leader}(n_c, n)$ and $\texttt{duration}(n_c, n)$. These functions take as input the round number n and the latest commit round known n_c to the node at the time they are invoked (Sections 7.5 and 7.7).

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

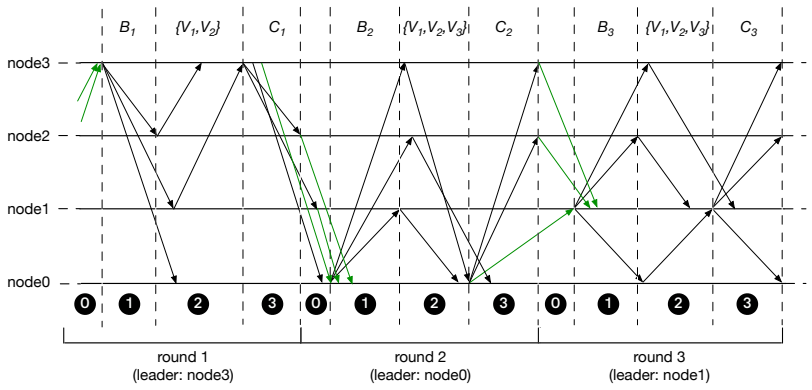

Figure 4: Successful rounds in LibraBFT (notifications to the new leader added in green)

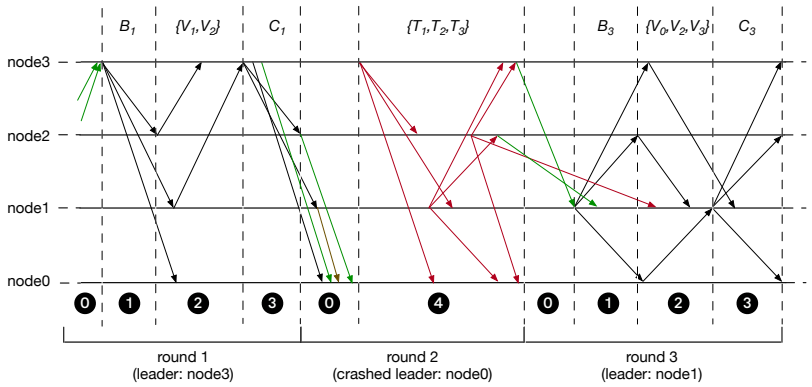

Figure 5: A failed round in LibraBFT (timeouts added in red)

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 (see Figure 5). New timeout objects are broadcast immediately (❹). 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.

Importantly, when a node enters a round, it must send its highest QC and TC to the new leader, in case the leader was not aware of the new data yet (❼ in Figure 4). A leader must re-share immediately its highest TC or QC when it enters its own round, together with a new proposed block (❶). Under GST assumptions, this guarantees that a leader node can propose not long after the first honest node entered its round. The fact that such a proposed block can provably meet the second voting constraint of each honest node is an important aspect of LibraBFT discussed in Section 8.3.

To keep communication complexity linear in the normal mode of operation, non-leader nodes may only trigger broadcast and query-all actions after some delay, in specific cases:

- A node executes a broadcast action whenever a round times out;
- After a round has timed out, a node triggers query-all actions periodically as long as its active round stays the same.
- A node also triggers query-all actions periodically when its highest known commit stays the same for too long.

The first mechanism lets nodes share their timeouts and form TCs in case the current leader is unresponsive. The second mechanism ensures that nodes eventually recover TCs and QCs that they may have missed due to malicious nodes or bad network conditions. Similarly, the third mechanism ensures that nodes eventually recover all commits. The set of records that sending nodes are required to re-share during network interactions is made precise in Section 7.12.

7.3. Pacemaker State and Update API

Visible pacemaker state. The pacemaker module is in charge of driving round synchronization. It exposes the following state values to the other components of a LibraBFT node:

- The current active epoch, denoted $\texttt{active\_epoch}(\alpha)$, (initial value: the first epoch identifier);
- The current active round, denoted $\texttt{active\_round}(\alpha)$, (initial value: 0);
- The leader of the current round, denoted $\texttt{active\_leader}(\alpha)$, (initial value: $\bot$).

These values are accessed by the method `proposed_block`, called in the main handler (Section 5.7). This ensures that a node α may only vote for a block B of the current active epoch that satisfies $\texttt{round}(B) = \texttt{active\_round}(\alpha)$ and $\texttt{author}(B) = \texttt{active\_leader}(\alpha)$.

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

Pacemaker update API. The pacemaker module exposes a method `update_pacemaker` meant to be called by the higher-level method `update_node`. This method refreshes the state of the pacemaker 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. In this case, the node should update $\texttt{latest\_voted\_round}(\alpha)$ so that no vote can be further created at the same round (see Section 8.3 for a justification).
- 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,
        // Current epoch.
        epoch_id: EpochId,
        // Known records.
        record_store: &RecordStore,
        // Local time of the latest query-all by us.
        latest_query_all: NodeTime,
        // Current local time.
        clock: NodeTime,
    ) -> PacemakerUpdateActions;

    /// Current active epoch, round, and leader.
    fn active_epoch(&self) -> EpochId;
    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 {
    /// Whether to propose a block and on top of which QC hash.
    should_propose_block: Option<QuorumCertificateHash>,
    /// Whether we should create a timeout object for the given round.
    should_create_timeout: Option<Round>,
    /// Whether we need to send our records to a subset of nodes.
    should_send: Vec<Author>,
    /// Whether we need to broadcast data to all other nodes.
    should_broadcast: bool,
    /// Whether we need to request data from all other nodes.
    should_query_all: bool,
    /// Time at which to call `update_pacemaker` again, at the latest.
    next_scheduled_update: NodeTime,
}
```

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,
        clock: NodeTime,
        smr_context: &mut SMRContext,
    ) -> NodeUpdateActions {
        let mut actions = NodeUpdateActions::new();
        actions.next_scheduled_update = pacemaker_actions.next_scheduled_update;
        actions.should_broadcast = pacemaker_actions.should_broadcast;
        actions.should_query_all = pacemaker_actions.should_query_all;
        actions.should_send = pacemaker_actions.should_send;
        if let Some(round) = pacemaker_actions.should_create_timeout {
            self.record_store.create_timeout(self.local_author, round, smr_context);
            // Prevent voting at a round for which we have created a timeout already.
            self.latest_voted_round.max_update(round);
        }
        if let Some(previous_qc_hash) = pacemaker_actions.should_propose_block {
            self.record_store.propose_block(self.local_author, previous_qc_hash, clock, smr_context);
        }
        actions
    }
}
```

7.4. Functions Based on the Highest Commit Round

We define the leader and the maximum duration of each round n as a function of n and some additional parameters that depend on the latest commit known to a node.

Given a commit round n_c, Lemma S2 shows that there is no ambiguity across honest nodes regarding the committed block B_c and the execution state $\texttt{state}(C_c)$ defined by $\texttt{round}(B_c) = n_c$ and $B_c \leftarrow C_c$.

In the following, we will write $f(n_c, n)$ whenever a function f depends on a round number $n > n_c + 2$ and possibly additional parameters deduced from the chain of committed blocks ending with B_c, as well as the corresponding execution state $\texttt{state}(C_c)$.

We note that such a function $f(n_c, n)$ cannot depend on the other fields of the quorum certificate C_c because there is no agreement on these values until the next block is committed.

7.5. Prerequisite: Assigning Leaders to Rounds

Given a suitable record store, a highest commit round n_c, and a round $n > n_c + 2$, we assume an algorithm $\texttt{leader}(n_c, n)$ to select the leader of round n.

The exact choice of a selection function $\texttt{leader}(n_c, n)$ may affect concrete latency bounds of the liveness theorem (Theorem L9) and is left for future work. As a starting point, one may require a

statistically fair selection in the sense that the frequency of each sequence of k leaders $(a_1, \ldots, a_k)$ is proportional to the product of the voting rights of each author a_i.

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

We also note that depending on n_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 $\texttt{leader}(n, n')$ $(n' > n + 2)$ points to a particular node once n is the highest commit round.

To mitigate both risks, we intend to use a verifiable random function (VRF) [40] in the future. If the certified block at round n_c contains some seed $s = \texttt{VRF}_{\texttt{author}(n_c)}(\texttt{epoch\_id} \,||\, \texttt{round}(n_c))$, then we may define $\texttt{leader}(n_c, n) = \texttt{author}(\texttt{PRF}_s(n) \mod N)$ where PRF stands for the implementation of a pseudo-random function.

7.6. Prerequisite: Target Commit Interval

We assume a time delay $I > 0$. When a node observes no new commits, we expect it to trigger a query-all action at least once per period I. The shorter the delay, the more responsive the protocol will be when a period of asynchronous network ends or when nodes disagree on the latest commit due to a dishonest leader.

7.7. Prerequisite: Assigning Durations to Rounds

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

- $\Delta(n_c)$ represents the amount of time available for the first block proposer on top of n_c.
- $\gamma(n_c)$ is the exponent used to increase the time for subsequent proposers.

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

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

We make the following observations:

- For the first round after the commit rule, $\texttt{duration}(n_c, n_c + 3) = \Delta(n_c)$.
- Since as $\Delta(n_c) > 0$ and $\gamma(n_c) > 0$, when n grows, $\texttt{duration}(n_c, n)$ keeps increasing and is not bounded.
- As far as theoretical liveness is concerned, duration coefficients could be fixed: $\Delta(n_c) = \Delta > 0$ and $\gamma(n_c) = \gamma > 0$ (for instance $\gamma = 2$). We adopt this simplified convention in the subsequent code examples. However, we expect practical performance to depend on more meaningful values.

7.8. Prerequisite: Periodic Query-all after Timeouts

Finally, in addition to the previous parameters, we assume that every node can compute a value $\lambda(n_c) > 0$. Once a round n has reached its maximal duration, in addition to creating a timeout object, a node will periodically execute a query-all action until it exits round n. The period of the query-all action is given by the formula $\texttt{period}(n_c, n) = \lambda(n_c) \cdot \texttt{duration}(n_c, n)$.

In practice, we may choose $\lambda(n_c) = \lambda < 1$. For instance, $\lambda = \frac{1}{2}$ means that the period of query-all actions is half of the maximal duration of the round.

7.9. Pacemaker State

We may now complete the specifications of the pacemaker state:

```
struct PacemakerState {
    /// Active epoch.
    active_epoch: EpochId,
    /// Active round.
    active_round: Round,
    /// Leader of the active round.
    active_leader: Option<Author>,
    /// Time at which we entered the round.
    active_round_start_time: NodeTime,
    /// Maximal duration of the current round.
    active_round_duration: Duration,
    /// Maximal duration of the first round after a commit rule.
    delta: Duration,
    /// Exponent to increase round durations.
    gamma: f64,
    /// Coefficient to control the frequency of query-all actions.
    lambda: f64,
}
```

For simplicity, we have omitted the potential dependency to n_c for the parameters $\Delta(n_c)$, $\gamma(n_c)$, $\lambda(n_c)$ mentioned previously.

7.10. Pacemaker Update Handler

We now describe the implementation of the pacemaker update function seen in Section 7.3 using Rust (Table 4).

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
  - $0 = \texttt{round}(h_{\texttt{init}})$.
- If we just entered a new epoch or $\texttt{active\_round}(\alpha)$ was changed above:
  - Start a timer to track the duration of the round;
  - Set the current active leader $\texttt{active\_leader}(\alpha)$ to $\texttt{leader}(n_c, \texttt{active\_round}(\alpha))$.
  - Set the current maximal duration $\texttt{active\_round\_duration}(\alpha)$ to $\texttt{duration}(n_c, \texttt{active\_round}(\alpha))$.
  - If we are not the leader, notify the new leader $\texttt{active\_leader}(\alpha)$.
- If we are leader (i.e. $\texttt{active\_leader}(\alpha)$ points to $\texttt{local\_author}(\alpha)$) and have not proposed yet, 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 the current active round has exceeded its maximum duration and we have not created a timeout yet, then request the creation of a timeout at round `active_round`(α) and request a broadcast.
- After timeout, if we have not changed the active round or realized a query-all action in an interval of time $\lambda(n_c) \cdot D$ — where D is the maximal duration of the current round used above — then request a new query-all action.
- Finally, reschedule a run of the main handler (hence this function) to the next time where we may have to create a timeout or request a query-all action.

7.11. Commit tracker

While the Pacemaker abstraction is responsible for driving the creation of blocks and QCs, the commit-tracker abstraction is used to track new commits and new epochs (Section 5.7). Specifically, the state of a commit tracker contains the latest processed epoch, the latest processed commit round, and the time of the latest processed commit.

The update function of the commit tracker aims to update its state as expected and enforce periodic query-all actions when no new commit is observed for a sufficiently long time (Section 7.6). We write `target_commit_interval` for the interval constant previously denoted I.

The logic of the commit tracker is described using Rust in Table 5.

7.12. Synchronizing 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 a sending node must re-share during a data-synchronization exchange.

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-timeouts)** If α_0 knows timeouts at its current active round, then α receives these timeouts.
>
> **(sync-block)** If α_0 proposed a block for the first time as a leader, then α receives this block and learns the chain of previous blocks and QCs required to verify this block.
>
> **(sync-vote)** If α_0 just voted for a block proposed by α as a leader, then α receives this vote.

```
fn update_pacemaker(
    &mut self,
    local_author: Author,
    epoch_id: EpochId,
    record_store: &RecordStore,
    latest_query_all_time: NodeTime,
    clock: NodeTime,
) -> PacemakerUpdateActions {
    // Initialize actions with default values.
    let mut actions = PacemakerUpdateActions::new();
    // Compute the active round from the current record store.
    let active_round = max(
        record_store.highest_quorum_certificate_round(),
        record_store.highest_timeout_certificate_round(),
    ) + 1;
    // If the epoch changed or the active round was just updated..
    if epoch_id > self.active_epoch
        || (epoch_id == self.active_epoch && active_round > self.active_round)
    {
        // .. store the new value
        self.active_epoch = epoch_id;
        self.active_round = active_round;
        // .. start a timer
        self.active_round_start_time = clock;
        // .. compute the leader
        self.active_leader = Some(Self::leader(record_store, active_round));
        // .. compute the duration
        self.active_round_duration = self.duration(record_store, active_round);
        // .. synchronize with the leader.
        if self.active_leader != Some(local_author) {
            actions.should_send = self.active_leader.into_iter().collect();
        }
    }
    // If we are the leader and have not proposed yet..
    if self.active_leader == Some(local_author) && 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());
        actions.should_broadcast = true;
        // .. force an immediate update to vote on our own proposal.
        actions.next_scheduled_update = clock;
    }
    if !record_store.has_timeout(local_author, active_round) {
        let timeout_deadline = self.active_round_start_time + self.active_round_duration;
        // If we have not created a timeout yet, check if the round has passed its maximal
        // duration. Then, either broadcast a new timeout now, or schedule an update
        // in the future.
        if clock >= timeout_deadline {
            actions.should_create_timeout = Some(active_round);
            actions.should_broadcast = true;
        } else {
            actions.next_scheduled_update =
                min(actions.next_scheduled_update, timeout_deadline);
        }
    } else {
        // Otherwise, enforce frequent query-all actions if we stay too long on the same round.
        let period = (self.lambda * self.active_round_duration as f64) as i64;
        let mut query_all_deadline = latest_query_all_time + period;
        if clock >= query_all_deadline {
            actions.should_query_all = true;
            query_all_deadline = clock + period;
        }
        actions.next_scheduled_update = min(actions.next_scheduled_update, query_all_deadline);
    }
    // Return all computed actions.
    actions
}
```

Table 4: Update function of the pacemaker

```rust
struct CommitTracker {
    /// Latest epoch identifier that was processed.
    epoch_id: EpochId,
    /// Round of the latest commit that was processed.
    highest_committed_round: Round,
    /// Time of the latest commit that was processed.
    latest_commit_time: NodeTime,
    /// Minimal interval between query-all actions when no commit happens.
    target_commit_interval: Duration,
}

struct CommitTrackerUpdateActions {
    /// Time at which to call `update_node` again, at the latest.
    next_scheduled_update: NodeTime,
    /// Whether we need to query all other nodes.
    should_query_all: bool,
}

impl CommitTracker {
    fn update_tracker(
        &mut self,
        latest_query_all_time: NodeTime,
        clock: NodeTime,
        current_epoch_id: EpochId,
        current_record_store: &RecordStore,
    ) -> CommitTrackerUpdateActions {
        let mut actions = CommitTrackerUpdateActions::new();
        // Update tracked values: epoch, round, and time of the latest commit.
        if current_epoch_id > self.epoch_id {
            self.epoch_id = current_epoch_id;
            self.highest_committed_round = current_record_store.highest_committed_round();
            self.latest_commit_time = clock;
        } else {
            let highest_committed_round = current_record_store.highest_committed_round();
            if highest_committed_round > self.highest_committed_round {
                self.highest_committed_round = highest_committed_round;
                self.latest_commit_time = clock;
            }
        }
        // Decide if too much time passed since the latest commit or the latest query-all action.
        let mut deadline = max(self.latest_commit_time, latest_query_all_time) + self.target_commit_interval;
        if clock >= deadline {
            // If yes, trigger a query-all action.
            actions.should_query_all = true;
            deadline = clock + self.target_commit_interval;
        }
        // Schedule the next update.
        actions.next_scheduled_update = deadline;
        // Return desired actions to main handler.
        actions
    }
}
```

Table 5: State, actions, and update function of the commit tracker

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. Commits from past epochs are delivered during data synchronization using the method `process_commits` introduced in Table 3. A solution for data synchronization is sketched 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. We sketch in-memory data-structures for the record store in Appendix A.2.

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 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 *maximum active round* to refer to the maximum active round between honest nodes at a given time. 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 maximum active round amongst honest nodes changes from n to n', then $n' = n + 1$.

PROOF: We have seen with Lemma L1 that the maximum active round cannot decrease. While the maximum 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 maximum active round increases sequentially is very important for the liveness argument. In particular, given that leaders are predictable until a new commit is produced, we must not allow malicious nodes and the network to cause a round to be skipped.
- Assuming that nodes agree on the highest commit round n_c, a similar argument as in the proof of Lemma L3 shows that the maximum active round n will stay the same for at least a time $\texttt{duration}(n_c, n)$ 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 maximum active round to change. We now aim at sufficient conditions for the *minimum active round* between honest nodes to increase. Note that by Lemma L1, the minimum active round never decreases.

Let n be the minimum active round at time t. We say that the system is *stabilized* at time t if the following conditions hold:

(i) There exists a round n_c and a time $t_c < t$ such that all honest nodes have had the same highest commit round n_c at least since t_c and until t.

(ii) Every honest node that entered an active round greater than n did it only after t_c.

(iii) Every honest node that created a timeout for its current active round n either did it after t_c, or has shared it at least once with every other honest node after t_c.

We start by showing that stabilization persists as long as no commit is produced after GST.

Proposition L4: Assume that all the honest nodes have the same highest commit round n_c at time $t > GST$ and at least until time $t' > t$. If the system is stabilized at time t, then it is stabilized at time t'.

PROOF: Conditions (i), (ii), and (iii) at time t' follows from the same condition at time t, Lemma L1, and the fact that $t' > t > t_c$. □

Let δ be the maximum time taken by a data-synchronization exchange between two nodes after GST (Section 2.5). Next, we provide sufficient conditions for a system to become stabilized.

Lemma L5: Assume that all the honest nodes have the same highest commit round n_c at time t and at least until time $t' = t + 2\,\delta$. Assume that $\texttt{leader}(n_c, n)$ is honest and $\texttt{duration}(n_c, n) > \delta$. If an honest node α first switches to a new maximum active round n at time t, then we have that:

- By time $t + \delta$, the leader of round n has entered its own round.
- By time $t' = t + 2\,\delta$, every node has received a new proposed block by the leader of round n, and the system is stabilized with minimum active round at least n.

PROOF: We prove that every honest node has an active round at least n at time t'. Since active rounds are always increasing (Lemma L1) and $t > GST$, conditions (i), (ii), and (iii) at time t' will immediately follow from the maximality of n, with $t_c = t$.

Given the definition of the method `update_pacemaker` (Section 7.10), α switches round at time t because it learned a QC or a TC at round $n - 1$, and it will immediately notify the leader of the new round. Under GST assumptions, by time $t + \delta$, the honest leader of round n will receive such a notification for the first time. When it does, the maximality of n and the constraint $\texttt{duration}(n_c, n) > \delta$ guarantees that no TC or QC at round n exists yet (see Lemma L3 and the following note). This means that between times t and $t + \delta$, the leader of round n will learn a QC or TC at round $n - 1$ and enter round n (exactly). By definition of the pacemaker (Section 7.10), the honest leader will broadcast a new proposed block at round n together with the QC or TC at round $n - 1$. This will bring all honest nodes to an active round at least n by time $t' = t + 2\,\delta$. □

Note: In the conditions of Lemma L5, if additionally $\texttt{duration}(n_c, n) > 2\,\delta$, we observe that no timeout at round n can be created between time t and $t + 2\,\delta$. Therefore, all honest nodes will switch exactly to the round n and proceed to vote on the proposal unless a new commit is produced or the leader of round n produces a QC before $t + 2\,\delta$.

Lemma L6: Assume that all the honest nodes have the same highest commit round n_c at time $t > GST$ and at least until time $t' = t + I + \delta$, where n is the maximum active round between honest nodes at time t. Then, at time t', the system is stabilized with a minimum active round at least n.

PROOF: By definition of the target commit interval I (Section 7.6) and given that the highest commit round n_c is shared and constant, every honest node must trigger a query-all action between times t and $t + I$. In particular, a node with the maximum active round n at time t will answer queries from every other honest node and provide them with a TC or a QC at round $n - 1$ before $t + I + \delta$. Similarly, any potential timeout object created before time t will be shared with every other honest node. We deduce that condition (i), (ii), and (iii) hold at time t' and that the minimum active round n_0 at time t' satisfies $n_0 \geq n$. □

Proposition L7: Assume that all the honest nodes have the same highest commit round n_c at time $t > GST$ and at least until time $t' = t + (1 + \lambda(n_c)) \cdot \texttt{duration}(n_c, n) + \delta$, where n is the minimum active round between honest nodes at time t. Then, at time t', the system is stabilized with a minimum active round at least $n + 1$.

PROOF: By Proposition L4, the system is stabilized at time t', thus we only need to prove the lower bound on the minimum active round.

Given the definition of the method `update_pacemaker` (Section 7.10), every honest node that stays at round n must

- create a timeout at round n after $\texttt{duration}(n_c, n)$ time if it has not already,
- after a timeout was created, execute a query-all communication step at least once per period $\texttt{period}(n_c, n) = \lambda(n_c)\,\texttt{duration}(n_c, n)$.

If an honest node has switched to a round greater than n by time $t + \texttt{duration}(n_c, n)$, then by time t', every node still at round n will have executed a query-all step and reached a round $n+1$ or greater.

Otherwise, we may assume that every honest node has an active round equal to n until time $t + \texttt{duration}(n_c, n)$. By this time, every honest node will have created a timeout object for round n. Given condition (iii), such timeout objects were either created after t_c and broadcast immediately, or created earlier but broadcast at least once after t_c. Given the GST assumptions, by time $t + \texttt{duration}(n_c, n) + \delta$, every honest node will have learned enough old and new timeouts to form a TC at round n, thus switching to the next active round $n + 1$ or greater. □

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 7.12), we deduce $n_0 \leq \texttt{round}(B)$. Therefore, α_0 can vote for B' according to the voting rule (locked-round). □

Application. In the formalism of LibraBFT, we deduce that a leader at round n may propose a block as soon as it enters the round n. Indeed, entering round n is caused by a QC at round $n-1$ or a TC at round $n-1$.

- In the case of QC at round $n-1$, we know that a quorum of nodes α (if honest) knew no QC higher than round $n-2$ at the times t_α where they voted for a block at round $n-1$.
- In the case of TC at round $n-1$, we know that a quorum of nodes α (if honest) had round $n-1$ and each of them had the highest QC round given by the field `highest_certified_block_round` of the timeout at time t_α where they created their timeout object. Since the leader at round n accepted their timeout object as valid (Section 4.2), it must know a QC at least as high as the field `highest_certified_block_round`.

In both cases, honest nodes will not vote again until they enter the round n, then receive a proposal from the leader (see the voting rules (increasing-round) and the processing of pacemaker updates Section 7.3). Therefore, Proposition L8 always apply whenever an honest leader proposes a block.

8.4. Main Proof

In the following, we use *commit round* to refer to the round of the highest committed block of a node. We define the *minimum commit round* and the *maximum commit round*, respectively, as the minimum and maximum commit rounds over all honest nodes at a given time. As before, we write δ for the maximum delay for one data-synchronization exchange after GST (Section 2.5).

Theorem L9 (Liveness): Let n_c be the minimum commit round at a time $t_0 > GST$ and n_0 be the maximum active round at time t_0. Let $n > n_0$ be such that $\texttt{leader}(n_c, n)$, $\texttt{leader}(n_c, n+1)$, and $\texttt{leader}(n_c, n+2)$ are honest and such that $\texttt{duration}(n_c, n) > 4\,\delta$. Then, the minimum commit round is greater than n_c at time

$$t = t_0 + 2\,I + (n - n_0 + 11)\,\delta + \sum_{k=n_0}^{n-1} (1 + \lambda(n_c)) \cdot \texttt{duration}(n_c, k)$$

Proof: Let $t' = t - I - \delta$. We note that $t' \geq t_0 > GST$. Assume that there exists a node α with an unchanged commit round n_c at time t. The policy on target commit interval (Section 7.6) entails that α executes at least one a query-all action between time t' and $t' + I = t - \delta$. If the highest commit round at time t' is higher than n_c, under GST assumptions, α will learn this commit round by time $t' + I + \delta = t$, contradiction.

Therefore, we may safely assume that all the honest nodes have the same commit round n_c from time t_0 to t'. During this interval, this means every honest node agrees on the leader and the duration of each round. By Lemma L6 and Proposition L4, the system is stabilized starting from time $t_1 = t_0 + I + \delta$ until t', and the minimum active round at time t_1 is at least n_0.

By applying Proposition L7 repeatedly, the minimum active round at time $t'_1 = t_1 + (n - n_0)\,\delta + \sum_{k=n_0}^{n-1} (1 + \lambda(n_c)) \cdot \texttt{duration}(n_c, k)$ is at least n. By Lemma L3, the maximum active rounds on and after t_0 follow the sequence of values n_0, $n_0 + 1$, $n_0 + 2$, etc. Therefore, there exists a time t_2 with $t_0 < t_2 \leq t'_1$ at which an honest node enters the new maximum active round n for the first time. By Lemma L5, the leader of round n will switch to its own round n between t_2 and $t_2 + \delta$.

No honest node will timeout until $t_2 + \texttt{duration}(n_c, n)$. The time available for the leader of round n to operate before a timeout may occur is at least $t_2 + \texttt{duration}(n_c, n) - (t_2 + \delta) = \texttt{duration}(n_c, n) - \delta > 3\,\delta$. This leaves enough time to complete the following expected tasks:

- Create a valid block at round n on top of the highest known QC;
- Broadcast it (time cost $\leq \delta$);
- Gather a quorum of votes (time cost $\leq \delta$) — the fact that honest nodes will vote for the proposed block is discussed below; and
- Complete a broadcast of the new QC, and notably have the next leader receive it (time cost $\leq \delta$).

Recall that $4\,\delta < \texttt{duration}(n_c, n) \leq \texttt{duration}(n_c, n+1) \leq \texttt{duration}(n_c, n+2)$. Therefore, the same reasoning applies to the next leaders at round $n+1$ and $n+2$. The leader of round $n+2$ will observe three QCs at contiguous rounds n, $n+1$, and $n+2$ – hence a commit – then broadcast. All honest nodes will have a commit round at least $n > n_c$ at time $t_3 = t_2 + (1 + 3 + 3 + 3)\,\delta = t_2 + 10\,\delta$.

Using previous equations, we have:

$$t_3 \leq t_0 + I + 11\,\delta + (n - n_0)\,\delta + \sum_{k=n_0}^{n-1} (1 + \lambda(n_c)) \cdot \texttt{duration}(n_c, k) = t - I \leq t$$

therefore the minimum commit round is greater than n_c at time t.

To conclude the proof, we now verify that honest nodes are able to vote consistently on the proposal of the leaders at round n, $n+1$, $n+2$:

- Honest nodes enter rounds n, $n+1$, $n+2$ after time $t_0 > GST$, and before time $t_3 - \delta$. Because $t_3 - \delta \leq t - I - \delta = t'$, when they do, by the previous assumption, their commit round is still n_c, therefore they all compute the same leaders.
- Because $4\,\delta < \texttt{duration}(n_c, n) \leq \texttt{duration}(n_c, n{+}1) \leq \texttt{duration}(n_c, n{+}2)$, no honest node will timeout at round n, $n{+}1$, $n{+}2$. (This would increase their `last_voted_round` by the definition of `process_pacemaker_actions` in Section 7.3.) Honest leaders propose a unique block at their round, therefore each of the three proposals will pass the first safety rule (increasing-round) of every honest node.
- Honest leaders propose only after receiving a QC or a TC at the previous round. They always reference their highest QC in their proposed block. Therefore, Proposition L8 on optimistic responsiveness shows that each proposal will also pass the second safety rule (locked-round).
- Execution is deterministic; therefore, all the honest node proposes the same execution state for each proposed block. □

Note: In practice, we expect the value I to be significantly larger than round timeouts so that the corresponding query-all actions are rarely triggered. In the context of Theorem L9, the definition of t includes a contribution $2\,I$ that can be mitigated as follows.

- For subsequent commits after GST, we may assume that the highest commit and the highest QC/TC known to honest nodes was just broadcast at time $t_0 > GST$. Lemma L6 is no longer needed, thus we may spare a term $I + \delta$.
- If the `leader` function is such that $\texttt{leader}(n_c, k) = \texttt{leader}(n'_c, k)$, $\lambda(n_c) = \lambda(n'_c)$, and $\texttt{duration}(n_c, k) = \texttt{duration}(n'_c, k)$ for every $n_c \leq n'_c \leq k \leq n + 2$, then in the course of the liveness proof, nodes are guaranteed to agree on the values of these functions disregarding their highest commit round n'_c. Therefore, a second term $I + \delta$ may also be spared.
- Note however that the formulation of Theorem L9 assumed for simplicity that no new epoch is triggered. From the point of view of the commit latency bound, epoch changes have the same effect as a new commit round n'_c such that $\texttt{leader}(n_c, n) \neq \texttt{leader}(n'_c, n)$. Hence, they always entail a term $I + \delta$, disregarding the choice of the leader function.

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 [39]. 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 $\texttt{leader}(n_c, n)$ for $\texttt{round}(B_0) < n < \texttt{round}(B)$ because there may not be agreement between consensus nodes on n_c, the highest commit round 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 a next version, we intend to provide experimental results using our simulation code [36] 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 analyze networking communication more precisely, with additional studies on message sizes and protocol optimizations. 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 6).

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 `committed_state` 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 7.

In the simulator used as a reference for this report, we implement the `RecordStore` APIs using the in-memory data structures described in Table 8. 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 simulator are described in Table 9. 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) -> Option<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, together with a commit certificate.
    fn commit(&mut self, state: &State, commit_certificate: Option<&QuorumCertificate>);

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

/// 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 epoch starting at a given state.
    fn configuration(&self, state: &State) -> EpochConfiguration;
}

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

Table 6: Programming interfaces for State Machine Replication

```
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 highest QC.
    fn highest_quorum_certificate(&self) -> Option<&QuorumCertificate>;
    /// Query the round of the highest TC.
    fn highest_timeout_certificate_round(&self) -> Round;
    /// Query the round of the highest commit.
    fn highest_committed_round(&self) -> Round;
    /// Query the last QC of the highest commit rule.
    fn highest_commit_certificate(&self) -> Option<&QuorumCertificate>;
    /// Current round as seen by the record store.
    fn current_round(&self) -> Round;

    /// Iterate on the committed blocks starting after the round `after_round` and ending with the
    /// highest commit known so far.
    fn committed_states_after(&self, after_round: Round) -> Vec<(Round, State)>;

    /// 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 and the second previous round of a block.
    fn previous_round(&self, block_hash: BlockHash) -> Round;
    fn second_previous_round(&self, block_hash: BlockHash) -> Round;
    /// 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 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 unknown_records(&self, known_qc_rounds: BTreeSet<Round>) -> Vec<Record>;
    fn insert_network_record(&mut self, record: Record, smr_context: &mut SMRContext);
}
```

Table 7: 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>,
    current_proposed_block: Option<BlockHash>,
    /// Computed round values.
    highest_quorum_certificate_round: Round,
    highest_quorum_certificate_hash: QuorumCertificateHash,
    highest_timeout_certificate_round: Round,
    current_round: Round,
    highest_committed_round: Round,
    highest_commit_certificate_hash: Option<QuorumCertificateHash>,
    /// 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 8: 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>,
}

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.
    /// Epochs older than the receiver's current epoch will be skipped, as well as chains
    /// of records ending with QC known to the receiver.
    records: Vec<(EpochId, Vec<Record>)>,
}
```

Table 9: 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] Calibra Research, "LibraBFT simulator." https://github.com/calibra/research.

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

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

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

[40] 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.