State Machine Replication in the Libra Blockchain

The LibraBFT Team

Abstract

This report describes the algorithmic core of LibraBFT, named LBFT, and discusses next steps in its production. The consensus protocol is responsible for ordering and finalizing transactions. LBFT decentralizes trust among a dynamically reconfigurable set of validators, and remains safe against network asynchrony and even if at any particular configuration *epoch*, a threshold of the participants are Byzantine.

LBFT 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. Several novel features distinguish LBFT from HotStuff. LBFT incorporates a novel round synchronization mechanism that provides bounded commit latency under synchrony. It introduces a nil-block vote that allows proposals to commit despite having faulty leaders. It encapsulates the correct behavior by participants in a separable trust zone, allowing it to run within a secure hardware enclave that reduces the attack surface on participants.

LBFT can reconfigure itself, by embedding configuration-change commands in the sequence. A new configuration epoch may change everything from the validator set to the protocol itself.

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 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 LBFT— the focus of this report — by which blockchain transactions are ordered and finalized. LBFT decentralizes trust among a set of validators that participate in the consensus protocol and maintain agreement on the history of transactions.

LBFT encompasses several important design considerations.

Permissioned. The first one is a strong commitment for maintaining the Libra blockchain decentralized. Initially, the participating validators will be permitted into the consensus network by an association consisting of a geographically distributed and diverse set of *Founding Members*, which are organizations chosen according to objective membership criteria with a vested interest in bootstrapping the Libra ecosystem. In the literature, this model is referred to as *permissioned*, an approach that promotes sustainability without wasting excessive computational power.

Over time, membership eligibility will gradually become open while preserving the guarantee of decentralization with careful governance. To facilitate dynamic membership, LBFT can reconfigure itself by embedding configuration-change commands that require special credentials in the sequence. A new configuration *epoch* may change every part of the algorithm, from the validator set to the protocol itself.

Byzantine Fault Tolerance. The second consideration is to prevent individual faults from contaminating the entire system. It is worth noting that generally, participants are expected to be credible and the network to have predictable transmission delays. LBFT is designed to mask any deviation from correct behavior in a third of the participants. These cover anything from a benign bit flipping in a node's storage to fully comprising a server by stealing its secret keys. Additionally, LBFT maintains consistency even during periods of unbounded communication delays or network disruptions. This reflects our belief that consensus protocols whose safety rely on synchrony would be inherently both complex and vulnerable to Denial-of-Service (DoS) attacks on the network.

High Assurance. The core logic of LBFT allows simple and robust implementation, paralleling that of public blockchains based on Nakamoto consensus [23]. Notably, the protocol is organized around a single communication phase and allows a concise safety argument and a robust implementation. LBFT thus bridges between the simplicity and modularity of public blockchains based on Nakamoto consensus, but decentralizes trust based on permissions.

LBFT is a cutting edge technology that opens new possibilities in BFT scalability. It is based on the HotStuff algorithm [26] whose key benefit is *consensus-linearity*. Briefly, this means that under a wide range of settings, consensus decisions cost no more communication than to simply spread the decision to everyone.

LBFT has various features that distinguish it from HotStuff, some of which are left for future implementation and are discussed only briefly here, in Section 8. The core protocol rotates the leader at every block in the chain in order to provide fairness and symmetry, at no extra communication overhead. It implements a liveness and synchronization mechanism that maintains linearity under a broad set of conditions, as well as provides concrete latency bounds that are discussed formally in [24]. An enhancement to the voting rules allows commits to proceed despite faulty leaders, and planned enhancements will uphold consistency guarantees against higher corruption thresholds than traditional BFT approaches [21]. These features will be described in more detail in future documents.

2 Preliminaries

The goal of LBFT is to maintain a database of programmable resources with fault tolerance. At the core, the system is designed around a state-machine-replication (SMR) engine where participants form agreement on a sequence of transactions and apply them in sequence order deterministically to the replicated database.

LBFT is designed for a classical settings in which an initial system consists of a networked system of n participants. The initial set of members is determined when the system is bootstrapped. LBFT can reconfigure itself by embedding configuration-change commands in the sequence. A new configuration epoch may change everything from the validator set to the protocol itself. This allows the system to churn and allow open participation down the line.

To model failures, we define an *adversary* that controls the network delays and the behavior of up to a threshold of the participants. In the standard distributed computing model, there are three adversarial settings:

Synchronous. The synchronous model is the one in which the Byzantine consensus problem was originally conceived [25, 20]. In the synchronous model, there exists some known finite time bound Δ, such that the adversary can delay message delivery from an honest origin by at most Δ. 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.

Asynchronous. In the Asynchronous model, the adversary can delay message delivery by any finite amount of time even between honest parties. Note that, whereas there is no bound on the delay to message delivery, messages sent between honest parties must eventually be delivered. The celebrated FLP result [17] indicates that in asynchronous settings, any consensus solution that tolerates a single, crash failure must

have an infinite execution. In lieu of synchrony assumptions, randomized algorithms, pioneered by Ben-Or [3] guarantee progress with high probability. A line of research gradually improved the scalability of such algorithms, including [15, 22, 7, 1]. LBFT is not designed as a pure asynchronous solution, though in the future, it may incorporate various components of randomized solutions to thwart adaptive attacks.

Partial synchrony. The adversarial model which is assumed for LBFT is a hybrid between synchronous and asynchronous models called partial synchrony. It models practical settings in which the network goes through transient periods of asynchrony (e.g., under attack) and maintains synchrony the rest of the times.

A solution approach for partially synchronous settings introduced by Dwork et al. [13] separates safety (at all times) from liveness (during periods of synchrony). DLS 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. The DLS approach underlies most practical BFT works to date as well as the most successful reliability solutions in the industry, for example, the Google Chubbie lock service [5], Yahoo's ZooKeeper [19], etcd [14], Google's Spanner [11], Apache Cassandra [8] and others.

Formally, the partial synchrony model assumes a Δ transmission bound similar to synchronous networks, and a special event called GST (Global Stabilization Time) such that:

- GST eventually happens after some unknown finite time.
- Every message sent at time t must be delivered by time $\max\{t, GST\} + \Delta$.

An alternative definition for partial synchrony is to assume that there is some finite, but unknown, upper bound on message delivery [13].

Thresholds. There are several key impossibility and lower bounds that need to be taken into consideration. Under the asynchronous (and partially synchronous) model, Fischer et al. [16] showed that Byzantine consensus cannot be solved with more than one third faults, that is, at most f faults where $n = 3f + 1$. In this settings, a *quorum* consists of $n - f = 2f + 1$ validators. Under synchrony, Dolev and Reischuk showed in [12] an $\Omega(n^2)$ worst case communication complexity.

3 Technical Background

LBFT is based on a line of replication protocols in the partial synchrony model with Byzantine Fault Tolerance (BFT), e.g., [13, 9, 18, 4, 6, **?**, 10]. It embraces cutting-edge techniques from these works to support at scale a replicated database of programmable resources.

Round-by-round BFT solutions. The classical solutions for practical BFT replication share a common, fundamental approach. They operate in a *round by round* manner. In each round, there is a fixed mapping that designates a *leader* for the round (e.g., by taking the round modulo n, the number of participants). The leader role is to populate the network with a unique proposal for the round.

The leader is successful if it populates the network with its proposal before honest validators give up on the round and time out. In this case, honest validators participate in the protocol phases for the round. Many classical practical BFT solutions operate in two phases per round. In the first phase, a quorum of validators *certifies* a unique proposal, forming a *quorum certificate*, or a *QC*. In the second phase, a quorum of votes on a certified proposal drives a *commit* decision. The leaders of future rounds always wait for a quorum of validators to report about the highest QC they voted for. If a quorum of validators report that they did not vote for a any QC in a round r, then this proves that no proposal was committed at round r.

HotStuff is a three-phase BFT replication protocol. In HotStuff, the first and second phases of a round are similar to PBFT, but the result of the second phase is a certified certificate, or a QC-of-QC, rather then a commit decision. A commit decision is reached upon getting a quorum of votes on the QC-of-QC (a QC-of-QC-of-QC).

LBFT embraces the three-phase HotStuff paradigm which brings two benefits, *optimistic responsiveness* and *linearity*, that are not simultaneously achieved in two-phase solutions. Jointly, these two properties provide *consensus linearity* without sacrificing asynchrony: An honest leader can prove the safety of a proposal by referencing a single QC (from the highest round). HotStuff was adopted in several subsequent works, including PaLa [10], BFTree [2], and others.

Regardless of the number of phases, a round might not succeed in making progress due to a faulty leader or because validators enter the rounds at different times. The case of unsuccessful rounds is discussed below.

Chaining. LBFT borrows a *chaining* paradigm that has become popular in blockchain BFT protocols and popularized by HotStuff. In the chaining approach, the three phases for commitment are spread across rounds. More specifically, every phase is carried in a round and contains a new proposal. The leader of round k drives only a single phase of certification of its proposal. In the next round, $k+1$, a leader again drives a single phase of certification. This phase has multiple purposes. The $k+1$ sends its own $k+1$ proposal. However, it also piggybacks the QC for the k proposal. In this way, certifying at round $k+1$ generates a QC for $k+1$, and a QC-of-QC for k. In the third round, $k+2$, the k proposal can become committed, the $(k+1)$ proposal can obtain a QC-of-QC, and the $(k+2)$ can obtain a QC.

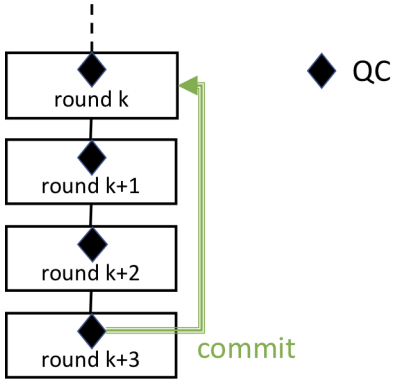

Figure 1: LBFT phases are spread across rounds.

Rounds without certificates. Validators may give up on a round by timeout. This may cause transition to the next round without obtaining a QC. The leader of the next round faces a choice as to how to extend the current block-tree it knows. LBFT deviates from HotStuff in that the leader always extend the highest certified leaf with a direct child. The advantage is that the tree of blocks has a uniform structure, every node has a QC for its direct parent.

Commit-rule. As a consequence of the LBFT chaining approach, the block tree in LBFT may contain "chains" that have gaps in round numbers. Round numbers are explicitly included in blocks, and the resulting commit logic is simple: It requires a 3-chain with contiguous round numbers whose last descendent has been certified.

Round synchronization. In a distributed system, at any moment in time, validators may be in a different state and receive different messages. The issue of bringing validators to enter rounds in approximately the same time is somewhat overlooked in the literature. PBFT gave a theoretical "eventuality" method for round synchronization by doubling the duration of rounds until progress is observed. HotStuff encapsulated the role of advancing rounds in a functionality named PaceMaker, but left its implementation unspecified. In LBFT, when a validator gives up on a certain round (say r), it broadcasts a *timeout* message carrying a certificate for entering the round. This brings all honest validators to r within the transmission delay bound

Δ. When timeout messages are collected from a quorum of validator, they form a *timeout certificate* (or a *TC*).

4 LBFT Overview

At a high level, the goal of the LBFT protocol is to commit blocks in sequence.

The protocol operates in a pipeline of *rounds*. In each round, a *leader* proposes a new block. Validators send their votes to the **leader of the next round**. When a quorum of votes is collected, the leader of the next round forms a *quorum certificate* (QC) and embeds it in the next proposal. This process continues until three uninterrupted leaders/rounds complete and the tail of a chain has three blocks with consecutive round numbers. Then, the head of the "3-chain" consisting of three consecutive rounds that has formed becomes committed. The entire branch ending with the newly committed block extends the sequence of commits.

Figure 2 depicts proposals (blocks), QC's, and a commit. The figure displays a *tree* of blocks, including a fork, rather than a simple chain. Forking can happen for various reasons such as a malicious leader, message losses, and others. For example, the figure shows a Byzantine leader that forked the chain at block k, causing an uncommitted chain to be abandoned. The k block uses the same QC for its parent as the left fork, hence validators vote it. In the depicted scenario, the k block becomes committed, and the left branch is discarded.

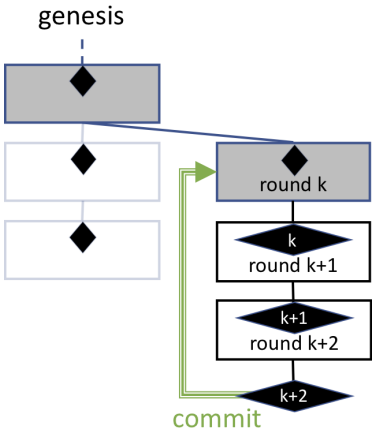

Figure 2: Proposals (blocks) pending in the Block-tree before and after a commit

LBFT guarantees that only one fork becomes committed through a simple voting rule that consists of two ingredients: First, validators vote in strictly increasing rounds. Second, whenever validators receive a block, they maintain a `preferred_round`, defined as the highest known grandparent round. The rule is that validators vote for a block if its parent round is as least the `preferred_round`. In Figure 2, validators that contributed to the formation of a QC for round $k+2$ remember k as their `preferred_round`.

Consider the scenario depicted in the Figure 3, which is intentionally quite tricky. In the scenario, QCs are formed in rounds $k+1$ and $k+2$. $2f+1$ validators vote for $k+2$ and remember k as their `preferred_round`. Then at round $k+3$ the leader experiences a temporary disruption. Round $k+3$ times out and another leader, say $k+4$ forks the chain at k. This fork obtains $2f+1$ votes because there is no violation of the `preferred_round` rule, and it is followed by $k+5$ and $k+6$.

In some future round, the leader of round $k+3$ becomes leader again and makes use of the QC for $k+2$, thus committing the round-k block. Later, another leader may make use of the QC for $k+6$ and hence commit $k+4$. Again, note that there is no consistency violation, the left-hand fork caused k to commit, and later it is dismissed.

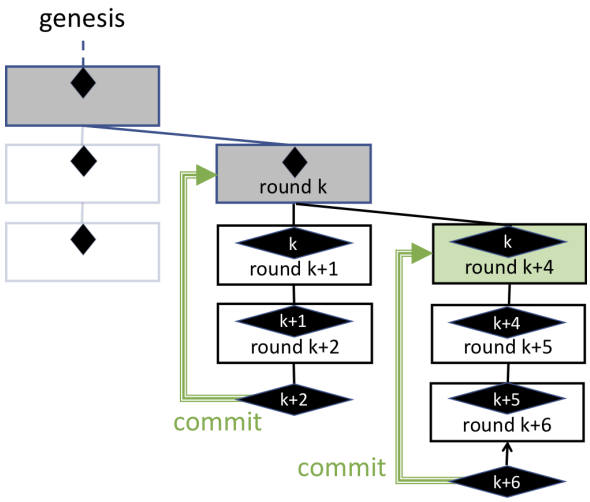

Figure 3: Third block of a three-chain does not necessarily commit

4.1 The Protocol in a Nutshell

The logical round-by-round protocol structure above is materialized in the LBFT codebase via a handful of event handlers executed by each validator. Handlers are triggered by either messages or timers.

More specifically, a validator main task is a loop `start_event_processing` handling three types of messages, a proposal (block), a vote, and a timeout. It is also triggered when a timer expires.

For brevity, the description henceforth assumes messages have already been filtered for format and validity of cryptographic values, and that any information indirectly referenced by messages (e.g., a QC for a block) is filled by a lower-level communication substrate.

Handling proposals, `process_proposal_msg`:

To bootstrap the execution, initially, all validators bootstrap the chain with a fixed genesis block and enter round 1. After the first round, validators enter rounds upon receiving a QC or TC for the previous round; vote and timeout collecting is described below.

Entering a new round is handled by `advance_round`. Upon entering a new round, the leader for the round extends the chain of blocks with a new proposal and broadcasts it.

When a validator receives a leader proposal for the current round k, it invokes `process_proposal_msg` to vote for the proposal according to the voting rules.

First, it processes the QC or TC for round $k-1$ the proposal carries, see details below (`process_certificates`). Subsequently, a validator invokes `make_vote` to determine if it can vote for the proposal according to two voting conditions:

1. A validator must vote in monotonically increasing rounds. Each validator maintains and persists a variable `last_vote_round`, and votes in round k if it is higher than `last_vote_round`. After it sends a vote, the validator last voting round is advanced to disallow voting again in round k (`increase_last_vote_round`).

2. When it votes on a block, a validator maintains and persists a variable `preferred_round`, and votes in round k only if the QC inside the k proposal is at least `preferred_round`.

Last, a validator prepares a speculatively committed state in case block k causes its grandparent to become committed. That is, if block k forms a 3-chain whose rounds numbers are consecutive to the grandparent,

then if a QC on k is collected the grandparent will be committed. Therefore, the validator includes in a vote a speculative commitable state `commit_state_id` of the grandparent.

Votes are sent to the leader of the next round $(k+1)$, along with the highest certificate (`high_qc`) known to the validator.

Handling votes, `process_vote_msg`:

When a validator receives a vote message for round k, it invokes `process_vote_msg` to handle it. First, it processes the `high_qc` the vote carries, see details below (`process_certificates`). Second, the Pacemaker aggregates the votes until it collects enough votes to form a QC. The Pacemaker authorizes a transition to the new via `advance_round`, causing the new leader of the new round to generate a proposed block (`generate_proposal`) and to broadcast it.

Handling timer expiration, `local_timeout_round`:

If a validator waits for the leader proposal for round k for a timeout period and it doesn't arrive, it broadcasts a timeout message carrying its highest certigicate `high_qc`.

Upon a timer expiration for round k, a validator broadcasts its timeout message to all the participants. A timeout message carries the highest certificate of the validator `high_qc`. Additionally, if the validator entered round k due to a TC, it includes the TC for round $k-1$ in the timeout message. Similarly to voting, the last voting round is advanced via `increase_last_vote_round` to disallow further voting in round k after expiring it.

Handling timeout message, `remote_timeout`:

When a validator receives a timeout message for round k, again it first processes the QC it carries via `process_certificates`, as described below. Second, the leader for round $k+1$ collects timeout messages for the preceding round until it forms a TC. Once a leader forms a TC, the leader generates a proposed block via `generate_proposal` and broadcasts it.

Processing certificates, `process_certificates`:

Upon receiving any message, a validator processes the certificates it carries via `process_certificates`:

Sync round: If a validator is currently at the certificate round or lower, it advances to the next round via `advance_round`.

Commit: If a certificate *qc* contains a non-empty commitable grandparent state `commit_state_id`, then the new committed state is executed and persisted to the ledger store.

Preferred round: The round of the parent of the certificate becomes "locked" by setting `preferred_round` to it, provided that it is higher than the current preferred round.

5 LBFT Full Protocol

We proceed with a detailed description of the LBFT protocol, elaborating the data-structures and modules in the implementation. The implementation is broken into the following modules: A `Ledger` module stores a local, forkable speculative ledger state. A `Block-tree` module creates proposal blocks and votes. It keeps track of a tree of blocks pending commitment with votes and QC's on them. A `Safety` module implements the core consensus safety rules. A `Pacemaker` module is the liveness keeper that advances rounds. A `ProposerElection` module maps rounds to leaders. Finally, a `Main` module is the glue that dispatches messages and timer event handlers.

Ledger

Ultimately, the goal of the Libra blockchain is to maintain a database of programmable resources, which the consensus LBFT core replicates. The database is represented by an abstract *ledger state*. Most of the implementation details of the persistent ledger-store and of the execution VM that applies transactions that mutate ledger state are intentionally left opaque and generic from the point of view of LBFT; in particular, the specifics of Move transaction execution are beyond the scope of this manuscript.

The `Ledger` module local to each LBFT validator serves as a gateway to the auxiliary ledger-store. It maintains a local pending (potentially branching) speculative ledger state that extends the last committed state. The speculation tree is kept local at the validator in-memory until one branch becomes committed. It provides a mapping from the block-tree (see below), which is pending commitment, to the speculative ledger state that pends commitment.

The `Ledger.speculate(prev_block_id, block_id, commands)` API speculatively executes a block of transactions over the previous block state and returns a new ledger state id. Speculative execution potentially branches the ledger state into multiple (conflicting) forks that form a tree of speculative states. Eventually, one branch becomes *committed* by the consensus engine. The `Ledger.commit()` API exports to the persistent ledger store a committed branch. Locally, it discards speculated branches that fork from ancestors of the committed state.

It is important to emphasize that Ledger supports speculative execution in order to enable proper handling of potential non-determinism in execution. If we would build a system that is merely ordering the commands in order to pass them to the execution layer, we would not need to maintain a tree of speculative states because the VM would execute the agreed commands only. However, such a system would not be able to tolerate any non-determinism (e.g., due to a hardware bug): the validators would diverge without LBFT being aware of that. Hence,LBFT goes beyond ordering the commands: it makes sure that the votes certify both the commands and their execution results. There should be at least $2f+1$ honest validators that arrive at the same state in order for the command to gather a QC.

```
Ledger
    new_state_id ← speculate(prev_block_id, block_id, cmds) // apply cmds speculatively
    state_id ← get_state(block_id) // find the pending state for the given block_id or nil if not
       present
    commit(block_id) // commit the pending prefix of the given block_id and prune other branches
```

Block-tree Module

The Block-tree module consists of two core data-types which the validator protocols is built around, blocks and votes. A third data-type derived from votes is a Quorum Certificate (QC), which consists of a set of votes on a block. An additional data-type concerned solely with timeouts and advancement of rounds is described below in the Pacemaker module. The Block-tree module keeps track of a tree of all blocks pending commitment and the votes they receive.

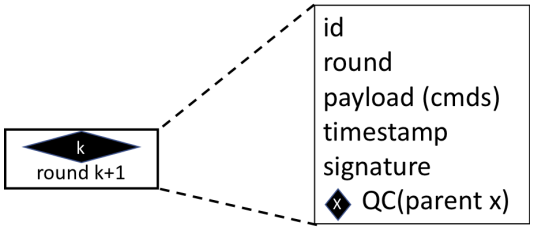

Figure 4: A block in `Block-tree`

Blocks. The core data structure used by the consensus protocol for forming agreement on ledger transactions is a `Block`. Each block contains as `payload` a set of proposed Ledger transactions, as well as additional information used for forming consensus decisions.

Every block `b` (except for a known genesis block P_0) is chained to a parent via `b.parent_qc`, a Quorum Certificate (QC) that consists of a quorum of votes for a parent block. In this way, the blocks pending commitment form a tree of proposed blocks rooted at P_0.

```
Block-tree
   Block
      round ; // the round that generated this proposal
      payload ; // proposed transaction(s)
      parent_qc ; // qc for parent block
      id ; // unique digest of round, payload and parent_qc.id
```

Votes and Certificates. Vote information `VoteInfo` for a block `b` must include both the block id `id` and an execution state id `exec_state_id` in order to guarantee a deterministic execution outcome. Additionally and (redundantly), `VoteInfo` holds the id's and rounds of `b`'s parent. This redundant information is kept for convenience, allowing to infer commitment from a single block without fetching its ancestors. This also allows to execute the core safety logic of LBFT within a TCB autonomously and determine commitment decisions without IO (see 8).

In addition, a vote message includes `LedgerCommitInfo`, a speculated committed ledger state, identified by `commit_state_id`. When a QC for a block results in a commit of its grandparent, the quorum of votes on the block also serve to certify the new committed ledger state.

`LedgerCommitInfo` serves two purposes. First, it includes the `commit_state_id`, which can be given to the clients as a proof of history (in practice `commit_state_id` can be a hash of a root of Merkle tree that covers the history of the ledger). Clients should not be aware of the specifics of consensus protocol for as long as they are able to verify that the given ledger state is signed by $2f+1$ participants. Second, `LedgerCommitInfo` includes the hash of `VoteInfo`. This hash is opaque to the clients and is used by Consensus participants. A validator that signs its vote message is thus authenticating both the potential `LedgerInfo` (to be stored by the ledger as a proof of commit) and the actual vote decision (to be used for running the Consensus protocol).

Consider, for example, a proposal `b` in round k with parent b' in round $k-1$ and grandparent b'' in round $k-2$. In case a validator decides to vote for `b`, it signs a `LedgerCommitInfo` that includes both the potential commit of b'' as well as the hash of `VoteInfo` on `b`.

```
Block-tree (cont.)
    VoteInfo
    |   id, round ; // id and round of block
    |   parent_id, parent_round; // id and round of parent
    |_  exec_state_id ; // speculated execution state

    // speculated new committed state to vote directly on
    LedgerCommitInfo
    |   commit_state_id ; // nil if no commit happens when this vote is aggregated to QC
    |_  vote_info_hash ; // hash of VoteMsg.vote_info

    VoteMsg
    |   vote_info ; // a VoteInfo record
    |   ledger_commit_info ; // Speculated ledger info
    |_  sender ← u, signature ← sign_u(ledger_commit_info);

    // QC is a VoteMsg with multiple signatures
    QC
    |   vote_info
    |   ledger_commit_info
    |_  signatures; // quorum of signatures
```

`PendingBlkTree` and `PendingVotes` The Block-tree module tracks blocks pending commitment in `PendingBlkTree` and votes in `PendingVotes`. The `PendingBlkTree` builds a speculative tree of blocks similarly to the `Ledger` building a speculative tree of states. In fact there is a 1:1 mapping between a block in `PendingBlkTree` and a block in `Ledger`. When a new block is added to the `PendingBlkTree` it is also automatically added to the `Ledger`.

Votes are aggregated in `PendingVotes` based on the hash of the `ledger_commit_info`. Once there are $2f + 1$ votes, they form a QC. As mentioned above, `ledger_commit_info` includes information both about a potential commit and `vote_info`, hence it is important to make sure that all vote messages refer to the same vote and commit (it would not be enough to aggregate votes just by the id of the proposal).

The algorithm maintains the highest known certified block in `high_qc`, updating it when a new QC is formed or received as part of the proposal. New proposals extend the highest certified block known locally to the validator.

Block-tree *(cont.)*

```
PendingBlkTree ; // tree of blocks pending commitment
PendingVotes ; // collected votes per block indexed by their LedgerInfo hash
high_qc ; // highest known QC

Procedure execute_and_insert(b)
    execute_state_id ← Ledger.speculate(P.parent_qc.block_id, P.id, P.payload)
    PendingBlkTree.add(b)
    high_qc ← max_round{b.parent_qc, high_qc}

Procedure process_vote(v)
    vote_idx ← hash(v.ledger_commit_info)
    V[vote_idx] ← V[vote_idx] ∪ v.signature
    if |V[vote_idx]| = 2f + 1 then
        qc ←QC⟨
            vote_info ← v.vote_info,
            state_id ← v.state_id,
            votes ← V[vote_idx]
            ⟩
        Pacemaker.advance_round(qc) high_qc ← max_round{qc, high_qc}

Function generate_proposal(cmds)
    return ⟨
        b.round ← current_round,
        b.payload ← cmds,
        b.parent_qc ← high_qc,
        b.id ← hash(b.round || b.payload || parent_qc.id)
    ⟩

Function process_commit(id)
    Ledger.commit(id)
    PendingBlkTree.prune(id) // id becomes the new root of pending
```

Safety Module

In LBFT, a block becomes committed when it becomes the head of a contiguous *3-chain* (three descendants with contiguous rounds). When a validator casts a vote on a block, it invokes Safety.`commit_rule` to check whether the vote may contribute to a new commit. A new commit will be enables if the qc inside the block immediately precedes the vote's round, and if the parent of the qc immediately precedes the qc round. In this case, the block referred to by `qc.parent_id` will become committed. Figure 2 demonstrates the application of the commit rule over a block-tree.

The consensus state per validator that is critical for the safety of the protocol consists of the two following counters: (i) `last_vote_round` keeps the last voted round, and (ii) `preferred_round` keeps the highest parent of a certified block (highest 2-chain head). Consensus state must be persisted before any vote or timeout message is sent to the peers.

Upon receiving a proposal (a block) $\mathtt{b}^*$, a validator votes for $\mathtt{b}^*$ only if $\mathtt{b}^*$.`round` is higher than its last voting round, and $\mathtt{b}^*$.`parent_round` is at least as high as `preferred_round`. Briefly, the first clause guarantees that at most one QC may be formed per round number. The second clause is slightly more tricky. When a validator votes on $\mathtt{b}^*$, there are already $2f + 1$ validators "locked" on the three-chain (great grandparent) round of $\mathtt{b}^*$ as their `preferred_round`. If indeed a QC on $\mathtt{b}^*$ is formed, then commitment of the great grandparent is safe. For a precise proof of safety, see Section 7.

The Safety module's logic is captured below.

```
Safety
    last_vote_round ; // initially 0
    preferred_round; // initially 0

    Procedure increase_last_vote_round(target)
        // commit not to vote in rounds lower than target
        if last_vote_round < target then last_vote_round ← target

    Procedure update_preferred_round(qc)
        preferred_round ← max{qc.parent_round, preferred_round}

    Function make_vote(block_id, block_round, parent_qc)
        // This function exercises both the voting and the commit rules
        if (block_round ≤ last_vote_round) ∨ (parent_qc.round < preferred_round) then
            return nil

        increase_last_vote_round(block_round)
        save_consensus_state()
        // VoteInfo carries the potential QC info with ids and rounds of the whole three-chain
        vote_info ←VoteInfo⟨
            (id, round) ← (block_id, block_round),
            (parent_id, parent_round) ← parent_qc.(id, round)
            (grandparent_id, grandparent_round) ← parent_qc.(parent_id, parent_round)
            exec_state_id ← Ledger.get_state(block_id)
            ⟩
        potential_commit_id ← commit_rule(parent_qc, block_round) ; // might be nil
        ledger_commit_info ←LedgerCommitInfo ⟨
            commit_state_id ← Ledger.get_state(potential_commit_id),
            vote_info_hash ← hash(vote_info)
            ⟩
        return VoteMsg⟨vote_info, ledger_commit_info⟩

    Function commit_rule(qc, vote_round)
        // find the committed id in case a qc is formed in the vote round
        if (qc.parent_round + 1 == qc.round) ∧ (qc.round + 1 == vote_round) then
            return qc.parent_id
        else
            return nil
```

Note that the safety module does not require external dependencies and generates votes coupled with the potential commit information purely based on the rounds carried by the proposal and its QC. This is helpful for verifying safety, as well as allowing to separate the Safety module to execute within a TCB (see 8).

Pacemaker Module

The advancement of rounds is governed by a module called Pacemaker. The Pacemaker keeps track of votes and of time.

In a "happy path", the Pacemaker module at each validator observes progress, a leader proposal for the current round, and advances to the next round.

In a "recovery path", the Pacemaker observes lack of progress in a round.

Upon a local round timeout, the Pacemaker broadcasts a `TimeoutMsg` notification.

Whenever the Pacemaker receives a certificate, either a quorum certificate (QC) or a timeout certificate (TC), it instructs the validator to advance to the certificate's round (Main.`process_new_round_event`). It

additionally instructs the validator to forward the QC/TC to the leader of round. This guarantees that upon entering a round with an honest leader, all validators will synchronize to within two network transmission delays.

```
Pacemaker
    current_round; // initially zero
    T[] ; // timeouts per round

    TimeoutMsg
        round ; highqc ← PendingBlkTree.get_high_qc() ; // sender's highest qc added automatically
        sender ← u, signature ← sign_u(round); // sender and signature added automatically

    Procedure local_timeout_round
        if u ∉ T[current_round] then
            Safety.increase_last_vote_round(current_round) // stop voting for current_round
            save_consensus_state()
            broadcast TimeoutMsg(current_round)
            T[current_round] ← T[current_round] ∪ {u}

    Procedure process_remote_timeout(tmo)
        T[tmo.round] ← T[tmo.round] ∪ {tmo.sender}
        // A timeout certificate (TC)
        if |T[tmo.round]| == 2f + 1 then advance_round(T[tmo.round])

    Procedure advance_round(qc)
        r ← qc.round
        if r < current_round then return
        stop local timer for round r
        current_round ← r + 1
        if u ≠ ProposerElection.get_leader(current_round) then
            send qc to ProposerElection.get_leader(current_round)
        start local timer for round current_round for duration get_round_timer(current_round)
        Main.process_new_round_event()

    Function get_round_timer(r)
        return round timer formula // for example, use 4 × Δ or α + β^commit_gap(r) if Δ is unknown.
```

ProposerElection Module

ProposerElection emits a validator for a given round in a deterministic fashion, s.t. different honest participants would all implicitly agree on the chosen leader per round. The leader is determined by the hash of the given round: as a result different combinations of successive leaders occur in the system with equal probability.

```
ProposerElection
    validators; // The list of current validators

    Function get_leader(round)
        return validators[ hash(round) mod |validators| ]
```

Main Module

The main body of LBFT is an event-handling switch that runs an endless loop, and invokes appropriate handlers to process messages and events. The following (handful) of high-level events are handled: a propose message, a vote message, a remote timeout message, a local timeout, and a new round (as leader).

```
Main
    loop: wait for next event M ; Main.start_event_processing(M)

    Procedure start_event_processing(M)
        if M is a proposal message then process_proposal_msg(M)
        if M is a vote message then process_vote_msg(M)
        if M is a timeout message then Pacemaker.process_remote_timeout(M)
        if M is a local timeout then Pacemaker.local_timeout_round ()

    Procedure process_certificates(qc)
        Pacemaker.advance_round(qc) // Might update current round
        Safety.update_preferred_round(qc)
        if qc.ledger_commit_info.commit_state_id ≠ nil then
            Block-tree.process_commit(qc.grandparent_id)

    Procedure process_proposal_msg(P)
        process_certificates(P.parent_qc) // Might update current round of pacemaker
        current_round ← Pacemaker.current_round
        if P.round ≠ current_round then
            return
        if P.sender ≠ ProposerElection.get_leader(current_round) then
            return
        Block-Tree.execute_and_insert(P) // Adds a new speculative state to the Ledger
        vote_msg ← Safety.make_vote(P.id, P.round, P.parent_qc)
        if vote_msg ≠ nil then
            vote_aggregator ← ProposerElection.get_leader(current_round + 1)
            send vote_msg to vote_aggregator

    Procedure process_vote_msg(M)
        Block-Tree.process_vote(M) // If a new QC is formed, Pacemaker will start a new round

    Procedure process_new_round_event()
        if u ≠ ProposerElection.get_leader(Pacemaker.current_round()) then return
        b ← Block-Tree.generate_proposal( new commands from mempool )
        broadcast sign_u(b)
```

Persisted Validator Information

In addition to the ledger state, which is abstracted away within the `Ledger` module, a validator needs to persist information about its voting history in order to maintain safety. The size of the state that needs to be persisted for safety is quite small, and consists of merely a few elements, persisted via `save_consensus_state()` as follows:

```
Procedure save_consensus_state()
    persist last_vote_round
    persist prefered_round
    persist PendingBlkTree // make voted information recoverable for liveness, not needed for safety
```

Omitted, gory details

We encapsulate several tasks related to wire-protocol and defer it to a *transport substrate* that will be described elsewhere. The transport takes care of formatting messages and serializing them for transmission over the wire, for reliably delivering messages, and for retrieving any data referenced by delivered messages, in particular, block ancestors. In particular, when a message is handled in the pseudo code, we assume that its format and signature has been validated, and that the receivers has synced up with all ancestors and any other data referenced by meta-information in the message.

6 Liveness Proof and Latency Bounds

6.1 Optimistic Responsiveness

We first prove *optimistic responsiveness*, namely that when an honest leader gets an opportunity to make progress – all honest validators wait for its proposal – they will accept and vote for it. In order to satisfy optimistic responsiveness, we need to demonstrate that at any round, if the leader is honest and a validator receives the leader proposal by the leader without the round timer expiring, then it accepts (votes) for the proposal.

Property 1. *If an honest validator has a* `preferred_round` $= r_0$ *when it enters round* r*, then there exist* $f + 1$ *honest validators, which had highest 1-chains at round* $r_{hqc} >= r_0$ *at round* $r' < r$*.*

Indeed, `preferred_round` $= r_0$ implies a 2-chain: there are $2f + 1$ validators that voted for a block carrying the QC with round r_0. This voting happened in some round $r' < r$, hence following our BFT assumptions there exist $f + 1$ honest validators with highest 1-chain equal or greater than r_0 before they enter round r.

In order to complete the optimistic responsiveness argument, consider an honest validator starting round r with preferred round r_0: by Property 1 some $f+1$ honest validators had 1-chains at least r_0 before entering r. Since the leader gathered the highest 1-chains from $2f + 1$ validators before entering r, there should be at least 1 honest validator that sent a 1-chain with round at least r_0 to the leader. Thus, a new proposal by the honest leader at round r carries a QC with round at least r_0, hence it passes the preferred round rule and should be accepted by this honest validators.

6.2 Liveness under Synchrony

We first analyze liveness/latency under synchrony. Assume a known bound Δ on messages transmission delays among honest validators, and let Pacemaker.`set_round_timeout`() return a fixed value 4Δ for all rounds.

Round synchronization. Under synchrony, the Pacemaker maintains *round synchronization* defined as follows:

Property 2. *If round* r *has an honest leader, then all honest validators enter round* r *within a period of* 2Δ *from each other.*

Indeed, an honest validator enters round r only upon receiving a QC containing $2f + 1$ round-$(r - 1)$ votes or a TC containing $2f + 1$ round-$(r - 1)$ timeouts. In the former case, the certificate is sent by the r leader, hence it arrives within Δ to all honest validators. In the latter, the first honest validator to receive a TC for round $(r - 1)$ forwards it to the (honest) leader, hence all remaining honest validators enter round r within 2Δ time.

A corollary of Property 2 is that all honest validators overlap in at least 2Δ period at round r.

By the corollary to Property 2, an honest leader and all honest validators overlap in at least 2Δ period in the leader round. Due to the optimistic responsiveness property, the honest validators will accept the leader proposal and vote for it.

We call a round *three-chainer* if it has an honest leader and three subsequent leaders are also honest. Applying the above two properties four times is succession, we get that the three-chainer proposal becomes committed.

We proceed to provide a concrete latency bound under synchrony. The latency to arrive at a new commit is impacted by two factor, the time to bring all honest validators to execute the same round after bad leaders, multiplied by the number of rounds to get a succession of four honest leaders.

Property 3. *If rounds $r, r+1, r+2, r+3$ have honest leaders, where r is the highest round any single honest validator has entered, then r becomes committed within $4 \times 6\Delta$ time.*

By Property 2, all honest validators enter round r within Δ period from each other. Furthermore, no honest validator round timer expires on any of the four rounds $r, r+1, r+2, r+3$, hence the round proposal commits. Since each round takes 3Δ, accounting for the additional 2Δ skew, the succession takes at most $4 \times (4\Delta + 2\Delta)$.

It is worth noting that the TC broadcast by honest validators is crucial for this latency bound. Without the broadcast, $f+1$ validators might advance jointly with f bad validator one round ahead of the remaining f honest, and the f honest never catch up.

The expected time e to arrive at an honest succession of four leaders varies and depends largely on the leader rotation regime, the number of actual faults, and their relation to the rotation scheme. For example, if f actual validator faults occur uniformly at random among n, $e \approx 12$.

In total, the expected latency is $(e \times 3\Delta) + (4 \times 6\Delta)$. Furthermore, suppose that δ is the **actual** network latency. We already showed that all honest validators enter the highest round any of them is it within Δ. Once they reach an honest 3-chain, it is easy to show (simply replace Δ above with δ) that is takes only 8δ for the 3-chain to become committed.

Note that the above latency bound is incurred in expectation on when the maximal number of faults is reached. Below, we elaborate on an enhancement called "nil blocks" that facilitates faster commits by individual honest leaders 8.

Liveness and Latency under Partial Synchrony.

What happens when Δ is not known (partial synchrony)? Pretty much the same except that validators need to "guess" Δ by gradually increasing their round duration.

Validators can choose round duration as $\beta + \alpha^{commit\_gap(r)}$, where $commit\_gap(r)$ is the number of rounds preceding r not known to commit minus 2 (since commits are always delayed two rounds). At any moment in time, different validators may have different views of the last commit, hence their round durations may vary. Eventually, all of them will have round durations at least Δ but the round timeout value is unbounded. That is, all honest validators will guarantee to enter new rounds within 2Δ delay, but the time they spend in a round until a timeout is unbounded. Therefore, the latency bound in a partial synchrony settings is the following property:

Again, once validators synchronize at a 3-chain round, they will progress to commit at network speed.

7 Safety Proof

In this section we start by defining some notation that we will use:

- $B_i \longleftarrow^* B_j$ means that the block B_j extends the block B_i.
- $B_i \longleftarrow QC_i \longleftarrow B_{i+1}$ means that the block B_i is certified by the quorum certificate QC_i which is contained in the block B_{i+1}.

- round(B_i) reads as "the round number of block B_i".
- previous_round(B_i) := round(B_{i-1}) for $B_{i-1} \longleftarrow QC_{i-1} \longleftarrow B_i$.
- preferred_round(N, B_i) reads as "the preferred round of validator N after voting for block B_i".

Let's now recapitulate the BFT assumption which we need to prove the safety of our protocol.

Definition 1 (BFT Assumption)**.** For all f dishonest validators, there exist $2f + 1$ honest validators.

The following classical lemma can be derived from the assumption.

Lemma 1. Under BFT assumption, for all blocks B, B' and quorum certificates QC, QC', if all the following properties are true

- $B \longleftarrow QC$
- $B' \longleftarrow QC'$
- $\text{round}(B) = \text{round}(B')$

then we have that $B = B'$. In other words there can only be one certified block per round.

Proof:

1. By definition, quorum certificates have at least $2f + 1$ votes. For a quorum certificate C let's define honest(C) as the set of honest validators that have participated in QC. Due to the *BFT assumption*, For all quorum certificate C we have

$$\text{honest}(C) \geq f + 1$$

2. Following *statement 1*, we have

$$|\text{honest}(QC) \cap \text{honest}(QC')| \geq (f + 1) + (f + 1) - (2f + 1) \geq 1$$

3. Following *statement 2*, There exists an honest node hn such that $hn \in QC$ and $hn \in QC'$. Since $\text{round}(B) = \text{round}(B')$, by *voting rule 1* hn could have only voted for one block in this round. This implies that $B = B'$.

We can now define what safety means formally.

Definition 2 (Safety)**.** LBFT is **safe** if $\forall\ B, \widetilde{B} \in Blocks$, such that B is committed at round k and $\widetilde{B}$ is committed at round $\widetilde{k} > k$, then $B \longleftarrow^* \widetilde{B}$.

Let's now state and prove the safety theorem.

Theorem 2. *LBFT is safe under BFT assumption.*

Proof:

1. Two blocks B_0, B_0' are committed if there exist two chains:
   - $B_0 \longleftarrow QC_{B_0} \longleftarrow B_1 \longleftarrow QC_{B_1} \longleftarrow B_2 \longleftarrow QC_{B_2}$
   - $\widetilde{B_0} \longleftarrow QC_{\widetilde{B_0}} \longleftarrow \widetilde{B_1} \longleftarrow QC_{\widetilde{B_1}} \longleftarrow \widetilde{B_2} \longleftarrow QC_{\widetilde{B_2}}$

   with both the chains having contiguous rounds, meaning that:
   - $\text{round}(B_2) = \text{round}(B_1) + 1 = \text{round}(B_0) + 2$
   - $\text{round}(\widetilde{B_2}) = \text{round}(\widetilde{B_1}) + 1 = \text{round}(\widetilde{B_0}) + 2$

We can assume round($\widetilde{B_0}$) $\geq$ round(B_0) without loss of generality.

2. **It suffices to prove** that $\forall$ B_0 and $\widetilde{B_0}$ such that round($\widetilde{B_0}$) $\geq$ round(B_0) and both blocks are committed, we have $B_0 \longleftarrow^* \widetilde{B_0}$.

3. Let B a certified block, using *lemma 1* the following statements are true:

- round(B) = round(B_0) $\implies$ $B = B_0$
- round(B) = round(B_1) $\implies$ $B = B_1$
- round(B) = round(B_2) $\implies$ $B = B_2$

4. Following *statement 2*, we have either of the **initial states**:

- round(B_0) $\leq$ round($\widetilde{B_0}$) $\leq$ round(B_2), and by *statement 3* $\widetilde{B_0}$ is either B_0, B_1, or B_2 and $B_0 \longleftarrow^* \widetilde{B_0}$,
- or round($\widetilde{B_0}$) > round(B_2).

5. Let block $\widetilde{B_i}$ such that round($\widetilde{B_i}$) > round(B_2) and let block $\widetilde{B_{i-1}}$ = previous_round($\widetilde{B_i}$), then either

- round(B_0) $\leq$ round($\widetilde{B_{i-1}}$) $\leq$ round(B_2), and by *statement 3* $\widetilde{B_{i-1}}$ is either B_0, B_1, or B_2 and $B_0 \longleftarrow^* \widetilde{B_{i-1}}$,
- or round($\widetilde{B_{i-1}}$) > round(B_2).

Proof:

5.1. It suffices to prove that if round($\widetilde{B_i}$) > round(B_2), then round($\widetilde{B_{i-1}}$) $\geq$ round(B_0).

5.2. Due to quorum certificates having at least $2f + 1$ votes, the intersection of any two quorum certificates has at least one honest validator.

5.3. By *statement 5.2*, There exists an honest validator hv such that $hv \in QC_{B_2}$ and $hv \in QC_{\widetilde{B_i}}$.

5.4. By the *voting rule 2*, after hv observed B_2, preferred_round(hv, B_2) $\geq$ round(B_0).

5.5. Since round($\widetilde{B_i}$) > round(B_2), by the *voting rule 1* hv could only vote for $\widetilde{B_i}$ if

previous_round($\widetilde{B_i}$) $\geq$ preferred_round(hv, B_2)
$\Leftrightarrow$ round($\widetilde{B_{i-1}}$) $\geq$ round(B_0).

6. $\forall$ blocks B_i, B_{i-1} such that B_{i-1} = previous_round(B_i), then round(B_{i-1}) < round(B_i).

7. By **induction** using *statements 4, 5 and 6*, $B_0 \longleftarrow^* \widetilde{B_0}$.

8 Next Steps

In this section, we briefly touch on some roadmap topics that LibraBFT will include in the future. More details about each topic will appear in coming documents.

Nil proposals. If a QC for a round (say k) cannot be formed, a leader of a higher round must present a timeout certificate (TC) from a quorum of validators in order to move into the round. If the TC contains timeout messages from validators, such that each of this quorum of validators did not vote in round k, then it can serve as a QC for an implicit "nil proposal".

The benefit of this enhancements is that expired rounds contribute to fast(er) commitment, as depicted in Figure 5. Note that nil proposals can be chained, just like normal blocks.

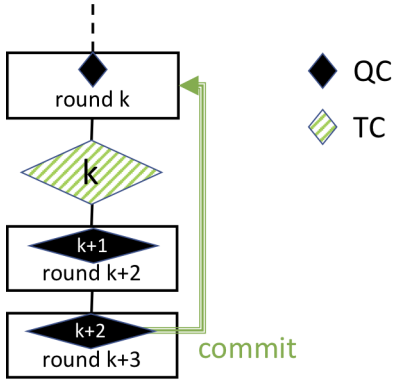

Figure 5: Nil blocks can contribute to fast(er) commitment.

Epoch changes. LBFT can reconfigure itself by embedding configuration-change commands in the sequence. Blocks on the branch extending a configuration-change command are proposed and voted on only for the purpose of commiting the reconfiguration, but they must not contain normal transaction payload (nor will transactions be they contain get executed). This is essentially a "stop the world and restart" operation.

A new configuration *epoch* may change every part of the algorithm, from the validator set to the protocol itself. A new epoch starts with an epoch-genesis block proposed and committed by the validators of the new epoch.

Proposer election strategies. There are several ways to enhance the proposer election mechanism in order to avoid performance hick-ups when a bad leader is elected, or worse, when a succession of bad leaders is elected.

An *alternate leader* strategy elects an ordered pair of leaders per round. The lower leader delays a certain duration in order to yield to the higher leader. This approach has the benefit of unlocking rounds with a crashed leader quickly, but the risk of creating contention among the leaders. The success of the approach largely hinges on the ability to stagger leaders effectively.

A different approach optimizes for a stable leader, and provides fairness and load balancing via rotating *input generators*. In each round, there are two distinct roles, a leader and an input generators. The leader is kept stable so long as progress and good performance are observed. The input generators are designated among the validators based on past performance and availability. In each round, the stable leader has the authority to promote or dismiss an input generator, but if it dismissed generators too frequently, the leader itself will get demoted eventually.

The last approach randomizes leaders using some source of randomness that cannot be predicted in advance.

TCB. As already mentioned in the Implementation section (Section 5), the `Safety` module of LBFT captures the rules that must be obeyed by a validator to guarantee agreement on committed proposals. First, the module succinct enough to allow formal verification of code implementations. Second, the interaction of the module with the environment is small: At bootstrap, the information that the validator persisted must

be uploaded with freshness proof to the safety module. Thereafter, the module can process proposed blocks autonomously and without any interaction with the environment. This allows to run the entire module within a secure, trusted compute base (TCB), provided that it is backed by a small, freshness-preserving persistent store.

Flexible BFT enhancements. The Flexible BFT [21] approach allows to support different commit assurance levels within the same protocol. There are several ways in which this approach may be harnessed to enhance the functionality of LBFT in the future. First, it can gradually increase assurance guarantees for past transactions, thus longer since a transaction has settled, the safer it becomes. This notion is similar to the gradually increasing finality of transactions in Bitcoin and Ethereum. Second, it allows to consider certain blocks as more critical, e.g., periodic checkpoints, high-stake transactions, and reconfiguration commands. Third, it supports adapting the commit parameters over time as evidence of corrupt/honest behavior accrues, and likewise, as experience is gathered on network transmission delays.

References

[1] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Validated asynchronous byzantine agreement with optimal resilience and asymptotically optimal time and word communication. In *38th ACM symposium on Principles of Distributed Computing (PODC'19)*, 2019. https://arxiv.org/abs/1811.01332.

[2] Jason Ansel and Marek Olszewski. Bftree - scaling hotstuff to millions of validators. *https://storage.googleapis.com/celo_whitepapers/*.

[3] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *2nd ACM symposium on Principles of Distributed Computing (PODC'83)*, pages 27–30, 1983.

[4] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *http://arxiv.org/abs/1807.04938v2*, 2018.

[5] Michael Burrows. The chubby lock service for loosely-coupled distributed systems. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 335–350, 2006.

[6] Vitalik Buterin and Virgil Griffith. Casper, the friendly finality gadget. *https://arxiv.org/abs/1710.09437*, 2017.

[7] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *25th Annual ACM Symposium on Theory of Computing (STOC'93)*, pages 42–51, 1993.

[8] Apache Cassandra. Apache cassandra. *Website, http://planetcassandra.org/what-is-apache-cassandra*.

[9] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *3rd symposium on Operating Systems Design and Implementation (OSDI'99)*, volume 99, pages 173–186, 1999.

[10] TH Hubert Chan, Rafael Pass, and Elaine Shi. Pala: A simple partially synchronous blockchain. https://eprint.iacr.org/2018/981, 2018.

[11] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, 2013.

[12] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *Journal of the ACM (JACM)*, 32(1):191–204, 1985.

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

[14] etcd community. etcd. *Website, https://etcd.io/*.

[15] Paul Feldman and Silvio Micali. Optimal algorithms for byzantine agreement. In *20th annual ACM symposium on Theory of Computing*, pages 148–161, 1988.

[16] Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, Mar 1986.

[17] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[18] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: a scalable decentralized trust infrastructure for blockchains. 2018.

[19] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*, 2010.

[20] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[21] Dahlia Malkhi, Kartik Nayak, and Ling Ren. Flexible byzantine fault tolerance. In *ACM CCS*, 2019.

[22] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *23rd ACM SIGSAC conference on Computer and Communications Security (CCS'16)*, pages 31–42, 2016.

[23] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *http://bitcoin.org/bitcoin.pdf*, 2008.

[24] Oded Naor, Mathiew Baudet, Dahlia Malkhi, and Alexander Spiegelman. Lumiere: Byzantine round synchronization. `https://arxiv.org/abs/1909.05204`, 2019.

[25] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[26] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *38th ACM symposium on Principles of Distributed Computing (PODC'19)*, 2019.