DiemBFT v4: State Machine Replication in the Diem Blockchain

The Diem Team [*]

Abstract

This report describes the 4^{th} version of the algorithmic core of Diem consensus, named DiemBFT, which is responsible for forming agreement on ordering and finalizing transactions among a configurable set of validators. The main goal of this report is to improve the latency of previous versions. We recognize that in the presence of a failed leader the view synchronization protocol of previous versions requires a quadratic number of messages. DiemBFT embrace this design choice and adopt a quadratic view-change of failed leaders, which enables it to commit blocks in two steps instead of three in the steady state. Additionally, DiemBFT incorporates a leader reputation mechanism that provides leader utilization under crash faults while making sure that not all committed blocks are proposed by Byzantine leaders. Leader utilization guarantees that crashed leaders are not elected, preventing unnecessary latency delays. Finally, DiemBFT formalizes the "safety isolation" guarantees. It encapsulates the correct behavior by participants in a "tcb"-able module of constant memory footprint, allowing it to run within a secure hardware enclave that reduces the attack surface on participants.

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 Diem 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 DiemBFT— the focus of this report — by which blockchain transactions are ordered and finalized. To facilitate agreement among all validator nodes on the ledger of transactions, the Diem Blockchain adopted the BFT approach by using the DiemBFT consensus protocol.

DiemBFT encompasses several important design considerations.

Permissioned & Open Network. The security of DiemBFT depends on the quality of validator node operators. Unlike prior work [14] that elects validator committees using Proof-of-Work, validator nodes in DiemBFT will be run by members of the Association or third-party operators approved by Diem Networks US to operate a validator node on their behalf. This model is referred to as *permissioned*, an approach that promotes security of the network based on the quality of participating Association Members and allows for sustainability without wasting excessive computational power.

[*]The authors work at Novi, a subsidiary of Facebook, Inc., and contribute this paper to the Diem Association under a Creative Commons Attribution 4.0 International License.

Revised August 18, 2021

Classical BFT. DiemBFT builds on a classical BFT approach pioneered by Lamport, Pease and Schostack in [15]. Four decades of scientific advances in this arena enable high transaction throughput, low latency, and a more energy-efficient approach to consensus than "proof of work" used in some other blockchains.

The main guarantee provided in this approach is resilience against Byzantine failures – preventing individual faults from contaminating the entire system. DiemBFT 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 compromising a server by stealing its secret keys. Additionally, DiemBFT maintains safety 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.

A second important guarantee this approach provides for DiemBFT is a clearly described *transaction finality* — when a participant sees confirmation of a transaction from a quorum of validators, they can be sure that the transaction has completed.

Enhancements. DiemBFT allows simple and robust implementation, paralleling that of public blockchains based on Nakamoto consensus [16]. Notably, the protocol is organized around a single communication phase and allows a concise safety argument and a robust implementation. DiemBFT thus bridges between the simplicity and modularity of public blockchains based on Nakamoto consensus, but builds trust based on the collective trustworthiness of the Association.

DiemBFT changes leaders in two separate cases. First, in normal operation DiemBFT rotates the leader-role among validators in order to provide fairness and symmetry. This does not require any extra communication. However, if done naively (e.g., round-robin), this may incur a big latency overhead since crashed leaders would keep being elected. In DiemBFT we design a novel leader election mechanism that achieves *leader utilization*. That is, the number of times a crashed leader is elected as leader is bounded. The leader election mechanism exploits the last committed state to implement a reputation scheme that tracks active validators. The tricky part is to agree on the last committed state. If done naively, a Byzantine adversary can make honest parties disagree on the leaders, which in turn may cause liveness or chain quality[1] violations.

Second, DiemBFT changes leaders when they are considered faulty. The previous version used the HotStuff linear view change mechanism to replace faulty leaders. However, since the view synchronization in previous versions requires quadratic communication anyway, DiemBFT uses a quadratic view-change mechanism that allows DiemBFT to reduce the HotStuff latency in the common case. In HotStuff terminology, DiemBFT has a 2-chain commit rule. This makes DiemBFT a hybrid between Hotstuff (common-case execution) and PBFT [6] (faulty leader execution). As a result during a stable network it has the linear cost of HotStuff, but during instability it pays a quadratic cost per round to a worse-case cubic cost similar to PBFT. As for the view synchronization, DiemBFT builds on top of the *time-out certificates* that was introduced in previous versions and adds a Bracha broadcast[1] inspired boosting mechanism for timeout messages to guarantee that all honest validators advance rounds in roughly the same speed [2]

Finally, validators in DiemBFT can verify safety in an isolated secure *safety module* by only storing locally a small (constant) amount of information. This enables the modeling of three types of validators: *honest*, *compromised*, and *Byzantine*, such that an adversary controls a compromised validator but cannot access its safety module. DiemBFT guarantees safety for any number of compromised validators as long as the faction of Byzantine ones is less than 1/3.

2 Problem Definition

The goal of DiemBFT is to maintain a database of programmable resources with fault tolerance. At the core, the system is designed around an SMR engine where validators form agreement on a sequence of transactions and apply them in sequence order deterministically to the replicated database.

[1]Percentage of blocks proposed by non-Byzantine leaders.
[2]Part of the 2-chain work appears in [11].

DiemBFT is designed for a classical settings in which an initial system consists of a networked system of n validators. The initial set of members is determined when the system is bootstrapped. To model failures, we define an *adversary* that controls the network delays and the behavior of some of the validators. We assume a secure trusted host and define three types of validators:

- Byzantine validator - all code is controlled by the adversary.
- Compromised validator - all code outside the secure trusted host is controlled by the adversary.
- Honest validator - nothing is controlled by the adversary

We collectively refer to honest and compromised validators as *non-Byzantine* validators and assume that all validators might be compromised, but there are at most $f < n/3$ Byzantine validators. The Agreement property requires that honest validators never commit contradicting chain prefixes. Note that compromised validators might locally commit anything since the committed information is stored outside the trusted hardware module. However, compromised validators cannot cause honest ones to violate agreement.

2.1 System Model

Network. The network model which is assumed for DiemBFT 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. [9] 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 [3], Yahoo's ZooKeeper [13], etcd [10], Google's Spanner [8], Apache Cassandra [5] 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 $\max\{t, GST\} + \Delta$.

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.

2.2 Technical Background

DiemBFT is based on a line of replication protocols in the partial synchrony model with Byzantine Fault Tolerance (BFT), e.g., [9, 6, 12, 2, 4, 14, 17, 7]. 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 and incur quadratic communication cost per decision (e.g., PBFT [6]). 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 any QC in a round r, then this proves that no proposal was committed at round r.

HotStuff, on the other hand, is a three-phase BFT replication protocol that has linear communication overhead in the common-case. 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).

DiemBFT is inspired by the linear three-phase HotStuff, but gets rid of the three-step latency cost. Instead, it preserves the communication linearity of HotStuff when the leader is alive but allows for a quadratic cost during the view-change protocol to regain the ability to commit in two steps.

Chaining. DiemBFT borrows a *chaining* paradigm that has become popular in blockchain BFT protocols. In the chaining approach, the 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 $k+1$ leader 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. As a result, in a 2-phase protocol, the k proposal can become committed, when the $(k+1)$ proposal obtains a QC.

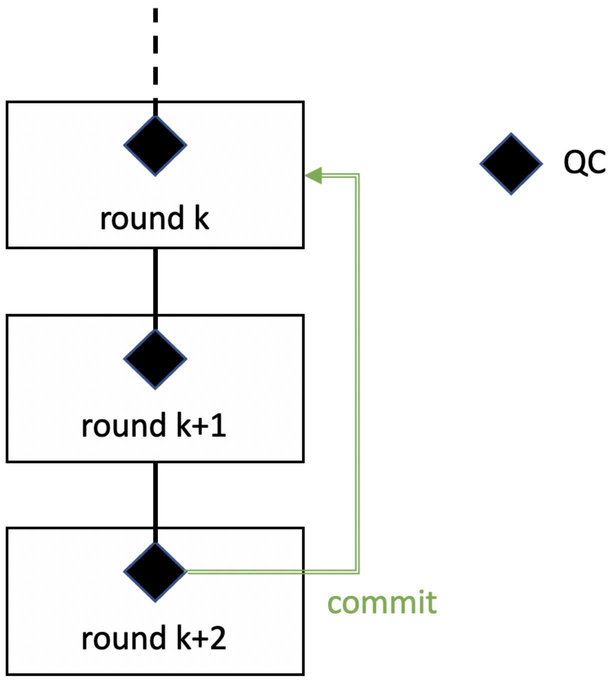

Figure 1: DiemBFT pipelines proposals and QC generation across rounds.

Round synchronization. In a distributed system, at any moment in time, validators may be in a different state and receive different messages. 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 DiemBFT, 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, they form a *timeout certificate* (*TC*). The TC also includes the signatures of the $2f+1$ nodes on the highest round QC they are aware of. This later serves as proof for the leader to safely extend the chain even if some parties are locked on a higher round than the one the newly elect leader. To make sure that, after GST, all validators are able to form TCs in a $O(\Delta)$ time for each other we use a Bracha [1] style mechanism to forward the timeout messages.

3 DiemBFT Protocol

The goal of the DiemBFT 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. Validators may also give up on a round by timeout. This may cause transition to the next round without obtaining a QC, in which case validators enter the next round through a view-change mechanism by forming or observing a *timeout certificate* (TC) of the current round. In fact, entering round $r+1$ requires observing either a QC or TC of round r. The leader of the next round faces a choice as to how to extend the current block-tree it knows. In DiemBFT the leader always extends the highest certified leaf with a direct child. The advantage is that the tree of blocks has a uniform structure and every node has a QC for its direct parent. As a consequence of the DiemBFT chaining approach, the block tree in DiemBFT 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 2-chain with* ***contiguous round numbers*** *whose last descendent has been certified.* When two uninterrupted rounds complete, the head of the "2-chain", consisting of the two consecutive rounds that have formed a QC, becomes committed. See illustration in Figure 1.

The entire branch ending with the newly committed block becomes committed. Figure 2 displays a *tree* of blocks, including an uncommitted fork. 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. In the depicted scenario, the k block becomes committed while the left branch is discarded. Note that, as we shortly explain, our safety rules guarantees that committed branches can never be discarded.

DiemBFT 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, each block has to include a QC or a TC from the previous round. If the previous round results in a TC then the validators check that the new leader's proposal is safe to extend. This check consists of looking at the TC which bears the $2f+1$ `highest_qc_round` (the highest QC included in a block the validator voted for) from distinct nodes. If the new proposal extends the highest of these `highest_qc_round`, this serves as proof that nothing from a round higher can even be committed (Otherwise at least one would have reported a higher `highest_qc_round`). Consider the scenario depicted in Figure 3, in which the $k+4$ QC is formed committing the block from round $k+3$. In future rounds it will be impossible to form a TC that would allow extending a lower QC than $k+3$.

We proceed with a detailed description of the DiemBFT protocol, elaborating the data-structures and modules in the implementation. The implementation is broken into the following modules:

- First, a Main (Section 3.1) module that is the glue, dispatching messages and timer event handlers.
- A Ledger (Section 3.2) module that stores a local, forkable speculative ledger state. It provides the interface for SMR service and to be connected to higher level logic (the execution).

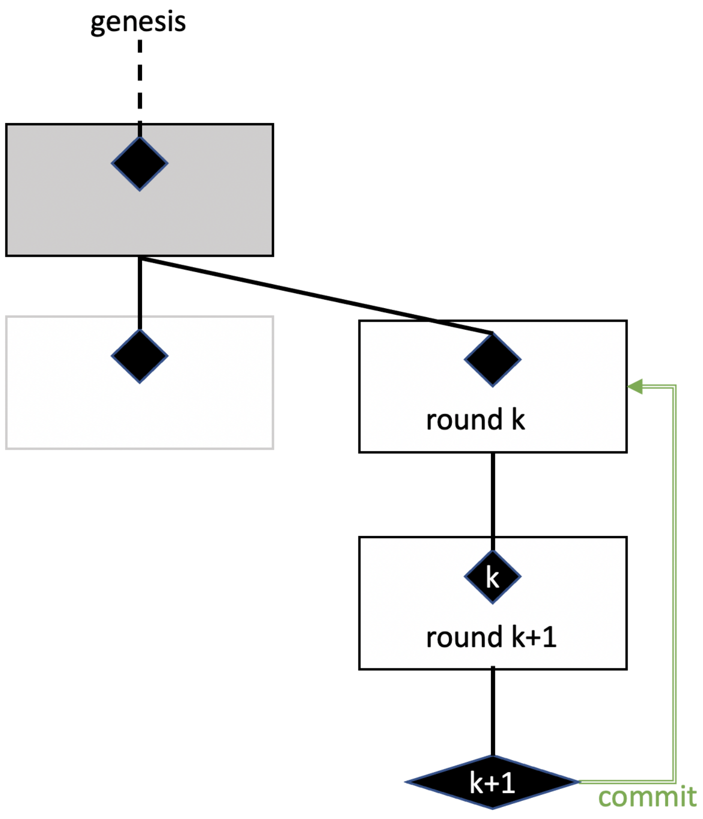

Figure 2: Proposals (blocks) pending in the Block-tree before and after a commit

- A Block-tree(Section 3.3) module that generates proposal blocks. It keeps track of a tree of blocks pending commitment with votes and QC's on them.
- A Safety (Section 3.4) module that implements the core consensus safety rules. The Safety : Private part controls the private key and handles the generation and verification of signatures. It maintains minimal state and can be protected by a secure hardware component.
- A Pacemaker (Section 3.5) module that maintains the liveness and advances rounds. It provides a "hearbeat" to Safety.
- A MemPool (Section 3.6) module that provides transactions to the leader when generating proposals.
- Finally, a LeaderElection (Section 3.7) module that maps rounds to leaders and achieves optimal leader utilization under static crash faults while maintaining chain quality under Byzantine faults.

Formal correctness proofs are given in Section 4.

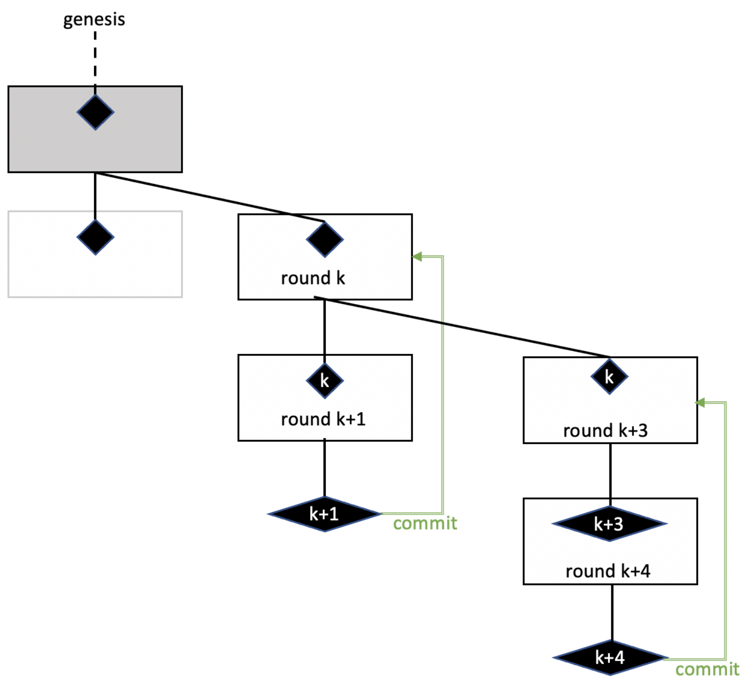

Figure 3: Committed blocks form a monotonically increasing chain.

3.1 Main Module

```
Main : EventLoop
    loop: wait for next event M ; Main.start_event_processing(M)
    Procedure start_event_processing(M)
        if M is a local timeout then Pacemaker.local_timeout_round()
        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 process_timeout_message(M)
```

The Main module of DiemBFT is an event-handling loop that invokes appropriate handlers to process messages and events. The following events are handled: a propose message, a vote message, a remote timeout message, and a local timeout.

```
Main
    Procedure process_certificate_qc(qc)
        Block-Tree.process_qc(qc)
        LeaderElection.update_leaders(qc)
        Pacemaker.advance_round(qc.vote_info.round)

    Procedure process_proposal_msg(P)
        process_certificate_qc(P.block.qc)
        process_certificate_qc(P.high_commit_qc)
        Pacemaker.advance_round_tc(P.last_round_tc)
        round ← Pacemaker.current_round
        leader ← LeaderElection.get_leader(current_round)
        if P.block.round ≠ round ∨ P.sender ≠ leader ∨ P.block.author ≠ leader then
            return
        Block-Tree.execute_and_insert(P) // Adds a new speculative state to the Ledger
        vote_msg ← Safety.make_vote(P.block, P.last_round_tc)
        if vote_msg ≠ ⊥ then
            send vote_msg to LeaderElection.get_leader(current_round + 1)

    Procedure process_timeout_msg(M)
        process_certificate_qc(M.tmo_info.high_qc)
        process_certificate_qc(M.high_commit_qc)
        Pacemaker.advance_round_tc(M.last_round_tc)
        tc ← Pacemaker.process_remote_timeout(M)
        if tc ≠ ⊥ then
            Pacemaker.advance_round(tc)
            process_new_round_event(tc)

    Procedure process_vote_msg(M)
        qc ← Block-Tree.process_vote(M)
        if qc ≠ ⊥ then
            process_certificate_qc(qc)
            process_new_round_event(⊥)

    Procedure process_new_round_event(last_tc)
        if u = LeaderElection.get_leader(Pacemaker.current_round) then
            // Leader code: generate proposal.
            b ← Block-Tree.generate_block( MemPool.get_transactions(),
             Pacemaker.current_round )
            broadcast ProposalMsg⟨b, last_tc, Block-Tree.high_commit_qc⟩
```

3.2 Ledger Module

Ultimately, the goal of the Diem blockchain is to maintain a database of programmable resources, which the consensus DiemBFT 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 DiemBFT; in particular, the specifics of Move transaction execution are beyond the scope of this manuscript.

The Ledger module, local to each DiemBFT 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 blocks that are pending commitment to the speculative ledger state.

The Ledger.`speculate`(`prev_block_id`, `block_id`, `txns`) 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 transactions 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 committed transactions 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 DiemBFT being aware of that. Hence, DiemBFT goes beyond ordering the transactions: it makes sure that the votes certify both the transactions and their execution results. There should be at least $2f + 1$ honest validators that arrive at the same state in order to form a QC for a block of transactions.

```
Ledger
    speculate(prev_block_id, block_id, txns) // apply txns speculatively
    pending_state(block_id) // find the pending state for the given block_id or ⊥ if not present
    commit(block_id) // commit the pending prefix of the given block_id and prune other branches
    committed_block(block_id) // returns a committed block given its id
```

DiemBFT only requires the above basic API from the Ledger module.

3.3 Block-tree Module

The Block-tree module consists of two core data-types which the validator protocols is built around, blocks and votes. Another 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.

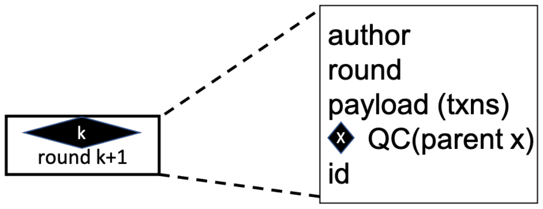

Figure 4: A block in `Block-tree`

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.qc`, a Quorum Certificate (QC) that consists of a quorum of votes for the parent block. In this way, the blocks pending commitment form a tree of proposed blocks rooted at P_0.

Vote information `VoteInfo` for a block `b` must include both the block `id` and the speculated execution state `exec_state_id` in order to guarantee a deterministic execution outcome. Additionally, `VoteInfo` holds the id's and rounds of `b`'s parent. This information is kept for convenience, allowing to infer commitment

```
Block-tree
    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; // ⊥ 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
        high_commit_qc; // QC to synchronize on committed blocks
        sender ← u; // Added automatically when constructed
        signature ← $\text{sign}_u$(ledger_commit_info); // Signed automatically when constructed

    // QC is a VoteMsg with multiple signatures
    QC
        vote_info;
        ledger_commit_info;
        signatures; // A quorum of signatures
        author ← u; // The validator that produced the qc
        author_signature ← $\text{sign}_u$(signatures);

    Block
        author; // The author of the block, may not be the same as qc.author after view-change
        round; // The round that generated this proposal
        payload ; // Proposed transaction(s)
        qc ; // QC for parent block
        id; // A unique digest of author, round, payload, qc.vote_info.id and qc.signatures
```

from a single block without fetching its ancestors. 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 parent, 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 need 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 `LedgerCommitInfo` (to be stored by the ledger as a proof of commit) and the QC embedded in the block it votes on (to be used for running the Consensus protocol). Note that `id` of the block includes the digest of signatures, so the set of voters for each committed round will be uniquely determined.

```
Block-Tree (cont.)
  pending_block_tree; // tree of blocks pending commitment
  pending_votes; // collected votes per block indexed by their LedgerInfo hash
  high_qc; // highest known QC
  high_commit_qc; // highest QC that serves as a commit certificate
  Procedure process_qc(qc)
    if qc.ledger_commit_info.commit_state_id ≠ ⊥ then
      Ledger.commit(qc.vote_info.parent_id)
      pending_block_tree.prune(qc.vote_info.parent_id) // parent id becomes the new root of
        pending
      high_commit_qc ← max_round{qc, high_commit_qc}
    high_qc ← max_round{qc, high_qc}
  Procedure execute_and_insert(b)
    Ledger.speculate(b.qc.block_id, b.id, b.payload)
    pending_block_tree.add(b)
  Function process_vote(v)
    process_qc(v.high_commit_qc)
    vote_idx ← hash(v.ledger_commit_info)
    pending_votes[vote_idx] ← pending_votes[vote_idx] ∪ v.signature
    if |pending_votes[vote_idx]| = 2f + 1 then
      qc ←QC ⟨
        vote_info ← v.vote_info,
        state_id ← v.state_id,
        votes ← pending_votes[vote_idx] ⟩
      return qc
    return ⊥
  Function generate_block(txns, current_round)
    return Block ⟨
      author ← u,
      round ← current_round,
      payload ← txns,
      qc ← high_qc,
      id ← hash(author || round || payload || qc.vote_info.id || qc.signatures) ⟩
```

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

The Block-Tree module tracks votes that have not formed a QC in pending_votes. The pending_block_tree is a speculative tree of blocks similar to the Ledger building a speculative tree of states. In fact, there is a 1:1 mapping between a block in pending_block_tree and a block in Ledger. When a new block is added to the pending_block_tree it is also 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.

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.

Remaining message and certificate types

There is another type of certificate - *timeout certificate*, used to advance a round when for some reason a QC on normal votes did not form. In addition to VoteMsg, there are two other types of messages TimeoutMsg and ProposalMsg.

```
TimeoutInfo
    round;
    high_qc;
    sender ← u; // Added automatically when constructed
    signature ← sign_u(round, high_qc.round); // Signed automatically when constructed

TC
    round; // All timeout messages that form TC have the same round
    tmo_high_qc_rounds; // A vector of $2f + 1$ high qc round numbers of timeout messages that form TC
    tmo_signatures; // A vector of $2f + 1$ validator signatures on (round, respective high qc round)

TimeoutMsg
    tmo_info; // TimeoutInfo for some round with a high_qc
    last_round_tc; // TC for tmo_info.round − 1 if tmo_info.high_qc.round ≠ tmo_info.round − 1, else ⊥
    high_commit_qc; // QC to synchronize on committed blocks

ProposalMsg
    block;
    last_round_tc; // TC for block.round − 1 if block.qc.vote_info.round ≠ block.round − 1, else ⊥
    high_commit_qc; // QC to synchronize on committed blocks
    signature ← sign_u(block.id);
```

A timeout or a proposal message for round r is *well-formed* if it contains the TC of round $r - 1$ as last_round_tc whenever the tmo_info.high_qc contained in the timeout message or block.qc (the parent QC) contained in the proposal message, respectively, aren't from round $r - 1$ (otherwise, last_round_tc is irrelevant and set to $\perp$ by convention). Messages that aren't well-formed are discarded by honest validators, and honest validators always generate well-formed timeout and proposal messages.

3.4 Safety Module

In DiemBFT, a block becomes committed when it becomes the head of a contiguous *2-chain*, i.e. the block round immediately follows the round of its parent block. This is checked in the commit_state_id_candidate function.

To be able to deploy the Safety module on a trusted hardware, DiemBFT maintains only the following two counters: (i) highest_vote_round keeps the last voted round, and (ii) highest_qc_round keeps the round of the highest QC included in a block the validator voted for.

Upon receiving a proposal (a block) `b`, a validator votes for `b` only if `b.round` is higher than its last voting round. Additionally, the validator evaluates the `safe_to_vote` predicate, which encapsulates the voting safety logic.

The full Safety module's logic is captured below. First we describe the private members and interfaces that can only be accessed from within the safety module.

```
Safety : Private
    private_key; // Own private key
    public_keys; // Public keys of all validators
    highest_vote_round; // initially 0
    highest_qc_round;

    Procedure increase_highest_vote_round(round)
        // commit not to vote in rounds lower than round
        highest_vote_round ← $\max\{$round, highest_vote_round$\}$

    Procedure update_highest_qc_round(qc_round)
        highest_qc_round ← $\max\{$qc_round, highest_qc_round$\}$

    Function consecutive(block_round, round)
        return round $+ 1 =$ block_round

    Function safe_to_extend(block_round, qc_round, tc)
        return consecutive(block_round, tc.round) $\wedge$ qc_round $\geq \max\{$tc.tmo_high_qc_rounds$\}$

    Function safe_to_vote(block_round, qc_round, tc)
        if block_round $\leq \max\{$highest_vote_round, qc_round$\}$ then
            // 1. must vote in monotonically increasing rounds
            // 2. must extend a smaller round
            return false
        // Extending qc from previous round or safe to extend due to tc
        return consecutive(block_round, qc_round) $\vee$ safe_to_extend(block_round, qc_round, tc)

    Function safe_to_timeout(round, qc_round, tc)
        if qc_round $<$ highest_qc_round $\vee$ round $\leq \max\{$highest_vote_round $- 1$, qc_round$\}$ then
            // respect highest_qc_round and don't timeout in a past round
            return false
        // qc or tc must allow entering the round to timeout
        return consecutive(round, qc_round) $\vee$ consecutive(round, tc.round)

    Function commit_state_id_candidate(block_round, qc)
        // find the committed id in case a qc is formed in the vote round
        if consecutive(block_round, qc.vote_info.round) then
            return Ledger.pending_state(qc.id)
        else
            return $\perp$
```

The public interface of the safety module described next is used by other modules to construct the two types of votes (`VoteMsg` and `TimeoutMsg`). Any valid vote is thus prepared and signed by the Safety module, which has the private keys. We also assume that `valid_signatures` call in the beginning of these functions checks the well-formedness and signatures on all parameters provided to construct the votes (using the public keys of other validators). Other parts of the system also check well-formedness and signatures (e.g. when receiving any type of message for the first time), but to protect against compromised validators the Safety needs to do its own layer of signature checking.

```
Safety : Public
    Function make_vote(b, last_tc)
        qc_round ← b.qc.vote_info.round
        if valid_signatures(b, last_tc) $\wedge$ safe_to_vote(b.round, qc_round, last_tc) then
            update_highest_qc_round(qc_round) // Protect qc round
            increase_highest_vote_round(b.round) // Don't vote again in this (or lower) round
            // VoteInfo carries the potential QC info with ids and rounds of the parent QC
            vote_info ←VoteInfo⟨
                (id, round) ← (b.id, b.round),
                (parent_id, parent_round) ← (b.qc.vote_info.id, qc_round)
                exec_state_id ← Ledger.pending_state(b.id) ⟩
            ledger_commit_info ←LedgerCommitInfo ⟨
                commit_state_id ← commit_state_id_candidate(b.round, b.qc),
                vote_info_hash ← hash(vote_info) ⟩
            return VoteMsg⟨vote_info, ledger_commit_info, Block-Tree.high_commit_qc⟩
        return $\perp$
    Function make_timeout(round, high_qc, last_tc)
        qc_round ← high_qc.vote_info.round;
        if valid_signatures(high_qc, last_tc) $\wedge$ safe_to_timeout(round, qc_round, last_tc) then
            increase_highest_vote_round(round) // Stop voting for round
            return TimeoutInfo⟨round, high_qc⟩
        return $\perp$
```

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.

3.5 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 uses the QCs in leader proposals to advance rounds. In a "recovery path", the Pacemaker observes lack of progress in a round and advances rounds based on timeout certificates.

Upon a local round timeout, the Pacemaker broadcasts a `TimeoutMsg` notification. This message contains `high_qc` and is signed by the Safety module, which verifies that the highest QC round is not higher than the round of `high_qc`. This is important for ensuring that later leaders will not be able to fork below the last committed block. Additionally, the `high_qc` information helps both the leader and the slow nodes to get up-to-date. After sending the `TimeoutMsg` message, validators increase their `highest_vote_round` to make sure they never vote in this round.

When a validator receives $f + 1$ timeout messages for round r, it timeouts round r if it has not done so already. When $2f + 1$ timeout messages are received a validator can form a TC and advance a round.

Therefore, if one has formed a TC, all other honest validators will do so within two network transition delays.

```
Pacemaker
    current_round; // Initially zero
    last_round_tc; // Initially ⊥
    pending_timeouts ; // Timeouts per round[a]
    Function get_round_timer(r)
        return round timer formula // For example, use $4 \times \Delta$ or $\alpha + \beta^{commit\_gap(r)}$ if $\Delta$ is unknown.
    Procedure start_timer(new_round)
        stop_timer(current_round)
        current_round ← new_round
        start local timer for round current_round for duration get_round_timer(current_round)
    Procedure local_timeout_round()
        save_consensus_state()
        timeout_info ← Safety.make_timeout(current_round, Block-Tree.high_qc, last_round_tc)
        broadcast TimeoutMsg⟨timeout_info, last_round_tc, Block-Tree.high_commit_qc⟩
    Function process_remote_timeout(tmo)
        tmo_info ← tmo.tmo_info
        if tmo_info.round < current_round then
            return ⊥
        if tmo_info.sender ∉ pending_timeouts[tmo_info.round].senders then
            pending_timeouts[tmo_info.round] ← pending_timeouts[tmo_info.round] ∪ {tmo_info}
        if |pending_timeouts[tmo_info.round].senders| == $f + 1$ then
            stop_timer(current_round)
            local_timeout_round() // Bracha timeout
        if |pending_timeouts[tmo_info.round].senders| == $2f + 1$ then
            return TC ⟨
                round ← tmo_info.round,
                tmo_high_qc_rounds ← {t.high_qc.round | t ∈ pending_timeouts[tmo_info.round]},
                signatures ← {t.signature | t ∈ pending_timeouts[tmo_info.round]}⟩)
        return ⊥
    Function advance_round_tc(tc)
        if tc = ⊥ ∨ tc.round < current_round then
            return false
        last_round_tc ← tc
        start_timer(tc.round + 1)
        return true
    Function advance_round_qc(qc)
        if qc.vote_info.round < current_round then
            return false
        last_round_tc ← ⊥
        start_timer(qc.vote_info.round + 1)
        return true
```

[a]Only need to store pending timeouts for one round - the current_round, which will also be the highest since any timeout message contains certificates that make the protocol enter the relevant round.

3.6 MemPool Abstract Module

We assume a MemPool module, which provides transactions to populate block payloads.

```
MemPool
    Function get_transactions()
```

3.7 LeaderElection Module

Classical consensus algorithms such as PBFT keep a stable leader until it is suspected to be faulty (crashed or Byzantine). This, however, is not a suitable strategy in blockchain protocols that need to ensure that proposals by honest validators are being committed, i.e., chain-quality. One solution would be to rotate the leader every round, giving every honest validator an equal chance to propose a value. However, this leads to crashed validators keep being elected as leaders without evidence they recovered.

To address this issue we propose a reputation-based leader election mechanism, taking prior active participation into account in determining eligibility to being elected a leader. However, if done naively such a mechanism could allow the adversary to control the selection of leaders or make honest validators disagree on leaders' identities.

```
LeaderElection
    validators; // The list of current validators
    window_size; // A parameter for the leader reputation algorithm
    exclude_size; // Between $f$ and $2f$, number of excluded authors of last committed blocks
    reputation_leaders; // Map from round numbers to leaders elected due to the reputation scheme
    Function elect_reputation_leader(qc)
        active_validators $\leftarrow \emptyset$ // validators that signed the last window_size committed blocks
        last_authors $\leftarrow \emptyset$ // ordered set of authors of last exclude_size committed blocks
        current_qc $\leftarrow$ qc
        for i = 0; i < window_size $\lor$ |last_authors| < exclude_size; i $\leftarrow$ i + 1 do
            current_block $\leftarrow$ Ledger.committed_block(current_qc.vote_info.parent_id)
            block_author $\leftarrow$ current_block.author
            if $i$ < window_size then
                active_validators $\leftarrow$ active_validators $\cup$ current_qc.signatures.signers()
                // |current_qc.signatures.signers()| $\geq 2f+1$
            if |last_authors| < exclude_size then
                last_authors $\leftarrow$ last_authors $\cup$ {block_author}
            current_qc $\leftarrow$ current_block.qc
        active_validators $\leftarrow$ active_validators $\setminus$ last_authors // contains at least 1 validator
        return active_validators.pick_one(seed $\leftarrow$ qc.voteinfo.round)
    Procedure update_leaders(qc)
        extended_round $\leftarrow$ qc.vote_info.parent_round
        qc_round $\leftarrow$ qc.vote_info.round
        current_round $\leftarrow$ PaceMaker.current_round
        if extended_round + 1 = qc_round $\land$ qc_round + 1 = current_round then
            reputation_leaders[current_round + 1] $\leftarrow$ elect_reputation_leader(qc)
    Function get_leader(round)
        if $\langle$round, leader$\rangle$ $\in$ reputation_leaders then
            return leader // Reputation-based leader
        return validators[$\lfloor \frac{\text{round}}{2} \rfloor \mod$ |validators|] // Round-robin leader (two rounds per leader)
```

In DiemBFT validators have two paths to determine the leaders. If an honest validator gets at round r a block with an embedded QC that commits round $r-2$, then the commit information is used by the leader reputation scheme to determine the leader of round $r+1$. In particular, the leader reputation scheme uses the QC signers information to determine the set of active validators, then (for chain quality purposes) excludes the `exclude_size` latest leaders of committed blocks, and deterministically choose a leader from the remaining set. Otherwise, if a validator does not commit round $r-2$ at round r (due to Byzantine behavior, crashes, or message delays), it uses a round-robin[3] fallback to determine the leader of round $r+1$. This means that different replicas may not follow the same path which would cause them to disagree on the next leader. This, however, can only happen a small number of times after GST as we prove later.

The purpose of the leader election mechanism is to reduce the impact of crash faults on the consensus's latency by detecting and excluding crashed validators from the leader rotation. Formally, for executions without Byzantine failures we require the following:

- **t-Leader-utilization:** Consider a run with no Byzantine validators. Then, after GST, there are at most t rounds in which honest validators do not agree on the leader or their leader is crashed.

This above property optimizes the more likely scenario in which leaders fail by crashing. However, the protocol must be resilient to worst case adversaries, including Byzantine validators. To this end, we require the following properties to hold under worst case Byzantine conditions:

- **Liveness:** An infinite number of blocks are committed after GST.
- **t-Chain-quality:** In any window of t committed blocks, at least one is proposed by an honest leader.

We prove the above guarantees as well as responsiveness bounds in Section 4.2.

In practice, we may want slow validators to also get a chance to become leaders. This can be accomplished by including validators not in `active_validators`$\cup$`last_authors` in consideration of being elected as leader with lower relative wight to their active counterparts. This introduces a trade-off with leader utilization.

4 Proof of Correctness

4.1 Agreement

We begin with some notation.

- We call a block Byzantine (honest) if it was proposed by a Byzantine (honest) validator.
- We say that a block B is *certified* if a quorum certificate $\texttt{QC}_B$ exists s.t. $B.\texttt{id} = \texttt{QC}_B.\texttt{VoteInfo.id}$.
- $B_i \longleftarrow \texttt{QC}_i \longleftarrow B_{i+1}$ means that the block B_i is certified by the quorum certificate $\texttt{QC}_i$ which is contained in the block B_{i+1}. We also use $B \longleftarrow \texttt{QC}_B \longleftarrow B'$ to express that $\texttt{QC}_B$ certifies block B and is extended by block B'.
- $B_i \longleftarrow^* B_j$ means that the block B_j *extends* the block B_i. That is, there is exists a sequence $B_i \longleftarrow \texttt{QC}_i \longleftarrow B_{i+1} \longleftarrow \texttt{QC}_{i+1} \cdots \longleftarrow \texttt{QC}_{j-1} \longleftarrow B_j$

Definition 1 (Global direct-commit)**.** *We say that a block B is* globally direct-committed *if $f+1$ non-Byzantine validators each call Safety.`make_vote` on block B' in round $B.\texttt{round}+1$, such that $B'.\texttt{QC}$ certifies B (i.e., $B'.\texttt{QC} = \texttt{QC}_B$), setting Safety.`highest_qc_round` $\leftarrow B.\texttt{round}$. These calls return $f+1$ matching votes (that could be used to form a $\texttt{QC}_{B'}$ with f other matching votes).*

[3]Note that for chain-quality purposes, every validator is considered twice in the round-robin.

Definition 2 (Local direct-commit)**.** *An honest validator locally direct-commits block B on the ledger by calling* Ledger.`commit(B.id)`*, only if it observes* $\texttt{QC}_{B'}$ *satisfying* $B \longleftarrow \texttt{QC}_B \longleftarrow B' \longleftarrow \texttt{QC}_{B'}$ *and* $B'.\texttt{round} = B.\texttt{round} + 1$*.*

Indeed, Ledger.`commit(B.id)` is called from Block-Tree.`process_qc`($\texttt{QC}_{B'}$). Moreover, we have that

Lemma 1. *An honest validator locally direct-commits block B on the ledger only if block B is globally direct-committed.*

Proof. By Definition 2, there exists a chain $B \longleftarrow \texttt{QC}_B \longleftarrow B' \longleftarrow \texttt{QC}_{B'}$ with $B'.\texttt{round} = B.\texttt{round} + 1$. The existence of $\texttt{QC}_{B'}$ implies that $f + 1$ non-Byzantine validators voted for B'. □

Our goal is to show that each committed block becomes part of the chain and can never be forked off. However, it may take time for honest validators to include B in the committed chain in the local ledger (how long it takes is analyzed in Section 4.2). This happens either after observing $\texttt{QC}_{B'}$ satisfying Definition 2, or by locally directly committing a block from a higher round - which as we prove below necessarily extends B.

Safety.`highest_vote_round` is updated every time a validator signs a timeout in Safety.`make_timeout` or signs a a vote in Safety.`make_vote`. A successful Safety.`make_vote` updates Safety.`highest_qc_round`. These variables are themselves used to determine whether to sign a vote or a time-out message, and we have

Lemma 2. *If a block B is globally direct-committed then any higher-round TC contains round at least* $B.\texttt{round}$ *within the* `tmo_high_qc_rounds`*.*

Proof. By Definition 1 $f + 1$ non-Byzantine validators call `update_high_qc_round` method at the beginning of Safety.`make_vote` on block `B'` at round $\texttt{B.round} + 1$, setting Safety.`highest_qc_round` $\leftarrow B.\texttt{round}$. None of these non-Byzantine validators may have previously executed Safety.`make_timeout` for round $B.\texttt{round} + 1$, as this would stop voting in this round (by invoking $\texttt{increase\_last\_vote\_round}(\texttt{B.round} + 1)$) making it impossible to prepare a vote on B' due to the $\texttt{block\_round} \leq \texttt{highest\_vote\_round}$ check in the `safe_to_vote` predicate. Finally, due to the checks in `safe_to_timeout`, a non-Byzantine validator can't prepare a timeout message for round $> B.\texttt{round}$ until it enters round $B.\texttt{round} + 1$.

Since Safety.`highest_qc_round` is never decreased, a timeout message by any of the above $f + 1$ non-Byzantine validators for a round $> B.\texttt{round}$ must be prepared by a Safety.`make_timeout` that observes $\texttt{high\_qc.vote\_info.round} \geq \texttt{highest\_qc\_round} \geq B.\texttt{round}$. By quorum intersection, timeout messages used to prepare the TC in any round $> B.\texttt{round}$ contain a message from one of these non-Byzantine validators, ensuring the corresponding round within `tmo_high_qc_rounds` is at least $B.\texttt{round}$. □

The next property follows from the quorum intersection, maximum number of Byzantine validators, the definition of compromised validators, and the fact that private keys are only stored in the Safety module:

Property 1. *If a block is certified in a round, no other block can gather $f + 1$ non-Byzantine votes in the same round. Hence, at most one block is certified in each round.*

Lemma 3. *For every certified block B' s.t.* $B'.\texttt{round} \geq B.\texttt{round}$ *such that B is globally direct-committed,* $B \longleftarrow^* B'$*.*

Proof. Let $r = B.\texttt{round}$. By Property 1, $B \longleftarrow^* B'$ for every B' s.t. $B'.\texttt{round} = r$.

We now prove the lemma by induction on the round numbers $r' > r$.

Base case: Let $r' = r + 1$. B is globally direct-committed, so by Definition 1, there are $f + 1$ non-Byzantine validators that prepare votes in round $r' = r + 1$ (in Safety.`make_vote`) on some block B_{r+1} such that $B \longleftarrow \texttt{QC}_B \longleftarrow B_{r+1}$. By Property 1, only B_{r+1} can be certified in round r'.

Step: We assume the Lemma holds up to round $r' - 1 > r$ and prove that it also holds for r'. If no block is certified at round r', then the induction step holds vacuously. Otherwise, let B' be a block certified in round r' and let $\texttt{QC}_{B'}$ be its certificate. B is globally direct-committed, so by Definition 1, there are $f+1$ non-Byzantine validators that have set Safety.`highest_qc_round` $\leftarrow r$ in their successful Safety.`make_vote` call in round $r+1$. One of these validators, v, must also have prepared a vote that is included in $\texttt{QC}_{B'}$ (as QC formation requires $2f+1$ votes and there are $3f+1$ total validators).

Let $B'' \longleftarrow \texttt{QC}_{B''} \longleftarrow B'$ and denote $r'' = B''.\texttt{round} = \texttt{QC}_{B''}.\texttt{vote\_info.round}$. There are two cases to consider, $r'' \geq r$ and $r'' < r$. In the first case, by the induction assumption for round $r'' < r'$ (blocks must extend smaller rounds to gather non-Byzantine votes and be certified), $B \longleftarrow^* B''$ and we are done.

In the second case, $r'' < r < r'$ (the right inequality is by the induction step), i.e., the rounds for B'' and B' are not consecutive. Hence, B' must contain a TC for round $r'-1$. By Lemma 2, this TC contains round $\geq r$ within `tmo_high_qc_rounds`. Consider a successful call by an non-Byzantine validator to Safety.`make_vote` for B'. The only way to satisfy the `safe_to_vote` predicate and vote for B' is by satisfying the `safe_to_extend` predicate, which implies $B''.\texttt{round} \geq \max(\texttt{tmo\_high\_qc\_rounds}) \geq r$, which is a contradiction to $r'' < r$. □

As a corollary of Lemma 3 and the fact that every globally direct-committed block is certified, we have

Property 2. *For every two globally direct-committed blocks B, B', either $B \longleftarrow^* B'$ or $B' \longleftarrow^* B$.*

For every locally committed block, there is a locally direct-committed block that extends it, and due to Lemma 1, also a globally direct-committed block that extends it. Each globally committed block defines a unique prefix to the genesis block, so Property 2 applies to all committed blocks.

4.2 Liveness

Lemma 4. *When an honest validator in round less than r receives a proposal or a timeout message for round r from another honest validator, it enters round r.*

Proof. Recall that the proposal and timeout messages sent by honest validators are well-formed, i.e. contain either a TC or QC of round $r-1$. When an honest validator receives and processes a well-formed timeout or proposal message, it calls PaceMaker.`advance_round_tc` on `last_round_tc`. If $\texttt{last\_round\_tc} \neq \bot$, then it is a TC from round $r-1$, and the validator enters round r.

Otherwise, `process_certificate_qc` call on `tmo_info.high_qc` (for timeout message) or `block.qc` (for proposal message), which internally calls PaceMaker.`advance_round_qc`, lets the validator enter round r. □

Recall that the leader election logic is implemented in LeaderElection.`update_leaders`. We refer to a round whose designated round-robin leader is honest as an *honest-round* and note that there are infinitely many honest-rounds. We refer to a leader whose proposal becomes certified at some round as the *elected-leader* of the round.

Lemma 5. *Suppose r is such that a QC forms in all rounds $> r$. Then, there are infinitely many rounds $> r$ with honest elected-leaders.*

Proof. Let $r' > r$ be an honest-round with (honest) designated round-robin leader ℓ. If all $2f+1$ honest validators send their round $r'-1$ votes to ℓ, then only ℓ can send round r proposal. The proposal must be sent, as otherwise a QC can't form for this round.

If some honest validator doesn't send its round $r'-1$ vote to ℓ, then a block B_{r-3} proposed in a round $r'-3$ must be committed. Since there are infinitely many honest-rounds higher than r, infinitely many blocks must be committed. As a corollary of chain quality (Section 4.4), eventually a honest block must be committed, implying that an honest leader sent the proposal. □

Lemma 6. *If the round timeouts and message delays between honest validators are finite, then all honest validators keep entering increasing rounds.*

Proof. Suppose all honest validators are in round r or above, and let v be an honest validator in round r.

We first prove that some honest validator enters round $r+1$. If all $2f+1$ honest validators time out in round r, then v will eventually receive $2f+1$ timeout messages, form a TC and enter round $r+1$. Otherwise, at least one honest validator, v' – not having sent a timeout message for round r – enters round $r+1$.

If a TC forms in any round $> r$, then a timeout message from some honest validator will eventually be delivered to v, triggering it to enter a higher round by Lemma 4. Similarly, if v' times-out in any round $> r$, then its timeout message will eventually trigger v to enter a higher round. Otherwise, v' must observe a QC in all rounds $> r$. In this case, by Lemma 5, an honest leader sends a proposal in some round $> r$. That proposal will eventually be delivered to v, triggering it to enter a higher round by Lemma 4. □

In an eventually synchronous setting, the system becomes synchronous after the the global stabilisation time (GST). Liveness in the classical sense follows from the following

Property 3. *Let r be a round after GST. Every honest validator eventually locally commits some block B by calling* Ledger.$\texttt{commit}(B.\texttt{id})$ *for* $B.\texttt{round} > r$.

Proof. By Lemma 6 honest nodes enter increasing rounds indefinitely.

We say that a round g is a *generating-round* if rounds $g, g+1, g+2$ are honest-rounds. We refer to them as the rounds *generated by* g. We make use of two generating-rounds $r' > r+3$ and $r'' \geq r'+3$.

If no honest validator uses the reputation-based scheme to determine the leader of any round generated by r', then all honest validators agree on honest-leaders for 3 consecutive rounds. In this case, they all locally direct-commit a block proposed in round r' – proven in Property 4 (next section) – and we are done.

Otherwise, some honest validator v' uses the reputation-scheme in one of the rounds generated by r'. In order to so, v' must have locally direct-committed a block B' in some round $\in [(r'-3), (r'-2), (r'-1)]$. If a QC never forms in some round $\geq r'+3$, then v' times-out and sends a timeout message, and the `high_commit_qc` in this message allows every receiver to locally commit block B' (or higher).

Thus, assume QCs form in rounds $\geq r''$ (since $r'' \geq r'+3$). If honest round-robin leaders form the QCs in honest-rounds generated by r'', then by Property 4, all honest validators will locally commit the block proposed in round r''. Otherwise, since QCs were formed in this round, $f+1$ honest validators must use the reputation-based scheme to determine a leader in one of the rounds generated by r'', implying that they each must have locally committed a block proposed in round $\geq r''-3 > r$. By Lemma 5, an honest leader ℓ sends a proposal in some $> r''+3$ round. The proposal must be based on a vote from at least one of the $f+1$ validators, and hence, contain the updated `high_commit_qc` that's as large as in the vote. Therefore, when the proposal is eventually delivered to all honest validators, it allows them to locally commit a block proposed in some round $> r$. □

4.3 Optimistic Time Bounds

We show that after GST, in an optimistic scenario when honest validators agree on the identity of leaders, the protocol makes progress at the network speed. We assume a known upper bound Δ on message transmission delays among honest validators (practically, a back-off mechanism can be used to estimate Δ) and let Pacemaker.`set_round_timeout`() return a fixed value 5Δ for all rounds. The results are given in terms of the worst-case tramission delay Δ, but the real time complexity of the protocol depends on the actual transmission delays (honest validators always just wait for sufficient messages to arrive to make progress).

Rounds are consecutive, advanced by quorum or timeout certificates, and honest validators wait for proposals in each round. We first show that honest validators that receive a proposal without the round timer expiring accept the proposal, allowing the quorum of honest validators to drive the system progress.

Lemma 7. *Let r be a round such that no QC has yet been formed for it and in which no honest validator has timed out. When an honest validator v invokes* `make_vote` *based on a proposal of an honest leader (according to v) of round r, it returns a vote message (not $\perp$).*

Proof. When the Safety module of an honest validator calls `safe_to_vote`, it checks that (1) round numbers are monotonically increasing and (2a) either `consecutive`(`block_round`, `qc_round`) holds, i.e. the block extends the QC of the previous round, or (2b) it's `safe_to_extend` based on the TC of the previous round.

Suppose `block_round` $= r$. For (1), by assumption none of the $2f+1$ honest validators have timed out, so no TC could have been formed for round r. Also by assumption, no QC has been formed. Hence, no honest validator may have entered or voted in a round larger than r. Round r has an honest leader, so when a honest validator calls `make_vote` for round r, it does so for the first time and with the largest voting round.

For (2), we consider two cases. If `last_round_tc` $= \perp$, then by well-formedness of honest leader's proposal, the `high_qc` it extends must have round number $r-1$, rounds are consecutive, and condition (a) holds.

If `last_round_tc` is not empty, then it is a TC for round $r-1$, formed based on $2f+1$ timeout messages. In the `safe_to_extend` predicate, the right-hand side is the maximum round among the `high_qc`'s in $2f+1$ timeout messages. The `qc_round` on the left-hand side is the round of the QC that the leader extended. Since the leader is honest, it is the `high_qc` of the leader's Block-Tree at the time when the proposal was generated. In this case, condition (b) holds (the left hand side of the `safe_to_extend` is not less than the right-hand side), because the honest leader updates the `high_qc` in its Block-Tree to have round at least as large as the `high_qc` of each timeout message it receives. □

We start by showing strong synchronization for certain types of rounds.

Lemma 8. *Suppose all honest validators agree that honest validator v is the round r leader and r occurs after GST. Then, all honest validators enter round r within a period of 2Δ from each other.*

Proof. When the first honest validator enters round r, if it is not the leader, it must have formed a TC for round $r-1$. Among the $2f+1$ validators whose timeout votes formed the TC, at least $f+1$ must be honest. These honest validators broadcast their timeout messages, and since it's after GST, every honest validator observes at least these $f+1$ timeout messages for round $r-1$ within time Δ. By Lemma 4, the first such timeout message triggers all honest validators that were in rounds less than $r-1$ to enter round $r-1$. Then, the Bracha timeout mechanism in PaceMaker.`process_remote_timeout` ensures that an honest validator in round $r-1$ immediately times out when it receives $f+1$ round $r-1$ timeouts. Thus, within 2Δ time, each honest validator still in round $r-1$ observes timeout messages for round $r-1$ from all honest parties, is able to form a TC of $2f+1$ timeout votes, and enters round r.

Either validator v is the first among honest validators to enter round r, or all honest validators (including v) enter round r during a 2Δ time after the first honest validator. If v is the first to enter round r, then it sends a proposal that arrives within time Δ to all honest validators that view v as the leader, by Lemma 4 triggering any that haven't yet entered round r to do so. □

Lemma 9. *If round r occurs after GST and all honest validators agree on honest leaders for rounds r and $r+1$, then the honest block proposed in round r is globally direct-committed within 5Δ time of the first honest validator entering round r.*

Proof. By Lemma 8, all honest validators receive round r proposal with block B_r from the leader within 3Δ time of starting their timer. By Lemma 7, honest validators accept the proposal and vote for it. Their votes are delivered to ℓ, the leader of round $r+1$ (whose identity they all agree on) within time Δ. ℓ will form a QC extending B_r and send a proposal with a block B_{r+1}. This proposal will be received by honest validators within another Δ time, by Lemma 4 triggering them to enter round $r+1$ before the local timer of 5Δ expires. By Lemma 7, every honest validator accepts the proposal and prepares a vote (vote is sent to whoever the validator believes to be the round $r+2$ leader). Since $f+1$ honest validators call Safety.`make_vote`$(B_{r+1}, \perp)$, by Definition 1 B_r is globally direct-committed. □

Finally, we show that all the committed blocks get locally committed by all honest validators. For this, honest validators must keep observing commit certificates $\texttt{QC}_{B_{r+1}}$ extending $\texttt{QC}_{B_r}$ for blocks B_r for ever increasing rounds r. This happens within tight time bounds in a pipelined fashion:

Property 4. *Suppose all honest validators agree on honest leaders of rounds $r, r+1, r+2$ and r occurs after GST. Then, every honest validator locally direct-commits the honest block proposed in round r within 7Δ time of when the first honest validator entered round r.*

Proof. We can continue the proof of Lemma 9, further knowing all honest validators agree that the leader of round $r+2$ is some honest validator v. v receives the votes to form the round $r+1$ QC after at most

6Δ time of the first honest validator entering round r. It then prepares and sends round $r+2$ proposal that includes and extends $\texttt{QC}_{B_{r+1}}$. After at most another Δ time (total 7Δ), all honest validators receive this proposal, leading them to locally direct-commit B_r. $\square$

4.4 Chain Quality

Here we study *chain quality*, i.e. the fraction of committed blocks that were proposed by honest validators. Note that chain quality is independent of protocol liveness. By liveness of the protocol, for any round r, eventually every honest validator locally commits a block from round $> r$, which by Property 2 uniquely determines the set of previously committed blocks. Therefore, in this section we don't need to distinguish between commit types (global, local, direct or not), as a block that is locally committed by any honest validator or is globally committed, is eventually locally committed by all honest validators.

We say that a honest validator v *considers* a validator ℓ to be the leader of some round r when v's call of LeaderElection.`get_leader(r)` returns ℓ. We call a validator ℓ a *reputation-based* leader of round r, if it could propose to $f+1$ honest validators that would consider ℓ the reputation-based leader of round r.

Lemma 10. *Let B be a committed block that was proposed by a Byzantine validator v in round r. If v is a reputation-based leader of round $r' \geq r+3$, then there are at least* `exclude_size` $- t + 1$ *honest committed blocks between r and r'.*

Proof. Honest validators that consider v as a reputation-based leader of round r' all have locally committed block B', proposed in round $r'-3 \geq r$. By Property 2, B' extends B. A validator in `last_authors` can't become a reputation-based leader, so v wasn't among the last `exclude_size` different authors of committed blocks going back from $b_{r'-3}$.

Hence, there are at least `exclude_size` blocks by different non$-v$ authors between B (exclusive) and B' (inclusive), thus between r and r'. There are at most t Byzantine validators including v, so at least `exclude_size` $-(t-1)$ of these blocks must be honest. $\square$

The next lemma easily follows from the design of the reputation mechanism.

Lemma 11. *Let round r be after GST such that rounds r and $r+1$ have the same honest leader according to the round-robin schedule. A block proposed in one of the rounds $r-3$, $r-2$ or r must be committed.*

Proof. If all honest validators used round-robin mechanism to determine the leader of rounds r and $r+1$ then Lemma 9 would apply and a block proposed in round r would be committed. Otherwise, a block proposed either in round $r-3$ or round $r-2$ must be committed (locally committed by an honest validator that used the reputation-scheme). $\square$

Lemma 12. *Suppose h honest blocks proposed in rounds $[r, r+6f+1]$ are committed, where r is an even round after GST. Then, there are $\geq 2f+1-3h$ committed Byzantine blocks proposed in rounds $[r-3, r+6f-2]$.*

Proof. Because r is even, the round-robin schedule over rounds $[r, r+6f+1]$ consists of $3f+1$ pairs of rounds $(r, r+1), \ldots, (r+6f, r+6f+1)$, each pair with the same, unique leader (in this schedule). $2f+1$ of these pairs have honest leaders according to the round-robin schedule. For each committed honest block that was proposed in round r', let us *disqualify* pairs that start with $2\lfloor r'/2 \rfloor$, $2\lfloor r'/2 \rfloor + 2$ and $2\lfloor r'/2 \rfloor + 4$, i.e. the pair containing r' and the following two pairs. Note that at most $3h$ pairs get disqualified, and if $2f+1 > 3h$, we can still find $d = 2f+1-3h$ honest validators whose pairs of rounds aren't disqualified. We establish a unique corresponding committed Byzantine block, proposed in rounds $[r-3, r+6f-2]$, for each of these honest validators, completing the argument.

Let v be one of the d honest validators whose pair $(r', r'+1)$ isn't disqualified - i.e. there are no honest committed blocks proposed in rounds $[r'-4, r'+1]$. By Lemma 11, a block proposed in rounds $r'-3$ or $r'-2$ has to be committed, and by the above, this block has to be Byzantine. It's left to show that for two different honest validators v_1 and v_2 with round-robin pairs (r_1, r_1+1) and (r_2, r_2+1), respectively, the sets $\{r_1-3, r_1-2\}$ and $\{r_2-3, r_2-2\}$ are disjoint – hence the corresponding Byzantine commits are unique.

$|r_1 - r_2| \leq 1$ holds if these sets intersect, which is impossible for distinct and even r_1 and r_2. As a result, there are at least d Byzantine committed blocks proposed in rounds $[r-3, r+6f-2]$. ☐

Theorem 13. *Let r be a round after GST. At least one honest block proposed among the next $O(f \log \frac{2f+1}{2f+1-2t})$ rounds gets committed. Moreover, at least* `exclude_size` $- t + 1$ *honest blocks proposed among the next $O(f(\log \frac{2f+1}{2f+1-2t} +$ `exclude_size` $- t + 1))$ rounds get committed.*

Proof. Without loss of generality, suppose r is even (at most one committed block per round). Starting from r, we consider phases of $6f + 6$ rounds and keep track of two variables: h_i, and d_i, which roughly track the number of honest (and Byzantine) committed blocks in a phase. Formally, we start by $h_0 = d_0 = 0$.

Let phase i start in round $r_i = r + (6f+6)i$. By Lemma 12 for round $r_i + 4$, if δ_h honest blocks proposed in rounds $[r_i + 4, r_i + 6f + 5]$ are committed, then there are at least $(2f + 1 - 3\delta_h)$ Byzantine committed blocks proposed in rounds $[r_i + 1, r_i + 6f + 2]$. Note the *buffer* of 4 rounds between round $r_i + 6f + 2$ and round $r_{i+1} = r_i + 6f + 6$ (start of the next phase).

Suppose δ_d among the Byzantine validators that proposed $(2f + 1 - 3\delta_h)$ committed Byzantine blocks didn't have any blocks committed in previous phases. We set $h_{i+1} = h_i + \delta_h$ and $d_{i+1} = d_i + \delta_d$.

Claim 1. *There are* `exclude_size` $- t + 1$ *honest commits within i phases, or $2d_i + 5\delta_d \geq 2f + 1 - 3\delta_h$.*

Proof. We can assume d_i Byzantine validators with blocks committed in previous phases don't become reputation-based leaders in rounds $[r_i + 1, r_i + 6f + 2]$. If they did, Lemma 10 would apply (due to the buffer, previously committed blocks are proposed in rounds $\leq r_{i-1} + 6f + 2 = r_i - 4$) and imply `exclude_size` $- t + 1$ honest commits. Hence, each of these validators propose at most twice.

We can also assume that in rounds $[r_i + 1, r_i + 6f + 2]$, any Byzantine validator proposes as a reputation-based leader at most 3 blocks that get committed. Otherwise, Lemma 10 again implies `exclude_size` $- t + 1$ honest commits (any four rounds contain two with numbers at least 3 apart). Each of the remaining δ_d Byzantine validators with committed blocks proposed in rounds $[r_i + 1, r_i + 6f + 2]$ may have at most 5 commits each, including ≤ 2 blocks proposed as a round-robin leader. ☐

In the following, we keep applying Claim 1. Since `exclude_size` $- t + 1$ honest commits immediately completes the proof of the theorem, assume $2d_i + 5\delta_d \geq 2f + 1 - 3\delta_h$ for the phases considered. Let's call phase i an H-phase if $\delta_h > 0$ and a D-phase if $\delta_h = 0$. Then, we have:

Claim 2. *After each D-phase, the quantity $2f + 1 - 2d_i$ decreases by at least a factor of $3/5$.*

Proof. In a D-phase, we have $\delta_h = 0$. Therefore, by Claim 1, we have $\delta_d \geq \frac{2f+1-2d_i}{5}$. Then,

$$\frac{2f + 1 - 2d_{i+1}}{2f + 1 - 2d_i} = 1 - \frac{2\delta_d}{2f + 1 - 2d_i} \leq 1 - 2/5 = 3/5.$$ ☐

By definition, $d_i \leq t$, implying $2f+1-d_i \geq 2f+1-2t$. Therefore, D-phase can occur at most $\log_{5/3} \frac{2f+1}{2f+1-2t} = O(\log \frac{2f+1}{2f+1-2t})$ times, which is also the maximum number of phases before the first H-phase. So, at least one honest block is committed in at most $O(\log \frac{2f+1}{2f+1-2t})$ phases, i.e. $O(f \log \frac{2f+1}{2f+1-2t})$ rounds.

For the second part of the theorem, note that the number of phases is the maximum number of D-phases plus `exclude_size` $- t + 1$. These contain at least `exclude_size` $- t + 1$ H-phases, implying the desired `exclude_size` $- t + 1$ honest commits. ☐

Corollary 1. *For $t = f$ and* `exclude_size` $= 2f$, *chain quality is $O(f \log f)$ commits per honest commit in the worst case, and $O(f)$ commits per honest commit on average.*

4.5 Leader Utilization

Leader utilization property is concerned with the protocol efficiency against crash failures. Therefore, we consider time after GST, and at most f validators that can crash, but aren't byzantine. We say that a validator crashes in round r if it has PaceMaker.`current_value` $= r$ at the time of the crash. Let t_r be the number of validators that crash in round r.

We call a validator *live* in a round r, if doesn't crash in a round $\leq r$. Let us define $p = \lceil \frac{3 \cdot \texttt{window\_size}}{2f+1} \rceil$.

Lemma 14. *Suppose $t_r > 0$ and round r occurs after GST. Define $r' = r + (p+1)(6f+6) - 1$ and let r'' be the smallest round number such that $r'' > r$ and $t(r'') > 0$, or ∞ if no such round exists. Then, rounds $r', \ldots r'' - 1$ all have live leaders that all non-crashed validators agree on.*

Proof. We assume $r'' > r'$, since otherwise there is nothing to prove.

As byzantine validators can be restricted to crash behavior, we can apply Lemma 12 p times: starting from round $2\lfloor r/2 \rfloor + 4$, then from round $2\lfloor r/2 \rfloor + 6f + 6$, etc, and finally from round $2\lfloor r/2 \rfloor + (p-1)(6f+6)$. We get that at least $p\lceil \frac{2f+1}{3} \rceil \geq$ `window_size` blocks proposed in rounds $r+1, \ldots (r' - 6f - 6)$ are committed. Validators that crashed in rounds $\leq r$ can't certify these blocks, so they are always excluded from `active_validators` when determining the reputation-based leader for rounds $(r' - 6f - 2), \ldots, r''$.

We show below that some round in $(r' - 6f - 3), \ldots, r'$ will have a reputation-based leader ℓ that knows it's the leader and that forms a QC on the votes from the previous round. There is no byzantine behavior, so ℓ knowing it's the reputation-based leader means its QC must extend the QC of the previous round, which must be a commit certificate for block proposed 3 rounds prior. Therefore, from this point until round r'', all validators will determine leaders by the reputation scheme. As we showed these reputation-based leaders are live. Due to agreement (Property 2) and the fact that the reputation-based leaders are determined based on committed blocks, validators never disagree on their identity.

At most f validators may crash, so by round-robin structure, we can find s with $r' - 6f - 3 \leq s \leq r' - 6$ such that the leaders of rounds $s, s+1, \ldots s+5$ according to the round-robin schedule are all live. If in all rounds $s+1, s+2, s+3$, a leader is allowed to collect votes on the proposal of the previous round and form a QC, then we would be done, because the round $s+3$ leader would have to be reputation-based. But if a QC never formed for (a proposal in) some round in $s, s+1, s+2$, then the next 3 rounds would necessarily have live round-robin based leaders that all validators will agree on. There will be progress by Property 4, triggering all the later leaders (until round r'') to be reputation-based (and live). $\square$

If the system has $t \leq f$ actual crash failures, the leader utilization property follows by applying Lemma 14 to at most t rounds with crash failures.

Corollary 2. *After GST, at most $(p+1)t \cdot (6f+6)$ rounds fail due to crash failures (i.e. crashed leader or validators disagreeing on the leader's identity).*

Further, we can treat the GST round like a round with a crash in the proof of Lemma 14 and obtain

Property 5. *The leader election algorithm satisfies $O(t \cdot \max(\texttt{window\_size}, f))$-Leader-utilization: after GST, for all but $(p+1)(t+1)(6f+6)$ rounds all validators agree on the leader that is live.*

5 Acknowledgements

We wish to thank Manuel Bravo, Gregory Chockler, Alexey Gotsman, Rachid Guerraoui, Kartik Nayak, Ling Ren and Maofan Yin for valuable feedback and discussions.

References

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

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

[3] 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.

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

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

[6] 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.

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

[8] 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.

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

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

[11] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. *arXiv preprint arXiv:2106.10362*, 2021.

[12] 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. In *DSN*, 2019.

[13] 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.

[14] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th {usenix} security symposium ({usenix} security 16)*, pages 279–296, 2016.

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

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

[17] 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.