AdoB: Bridging Benign and Byzantine Consensus with Atomic Distributed Objects[*]

WOLF HONORÉ[†], Yale University, USA
LONGFEI QIU, Yale University, USA
YOONSEUNG KIM, Yale University, USA
JI-YONG SHIN, Northeastern University, USA
JIEUNG KIM, Inha University, South Korea
ZHONG SHAO, Yale University, USA

Achieving consensus is a challenging and ubiquitous problem in distributed systems that is only made harder by the introduction of malicious byzantine servers. While significant effort has been devoted to the benign and byzantine failure models individually, no prior work has considered the mechanized verification of both in a generic way. We claim this is due to the lack of an appropriate abstraction that is capable of representing both benign and byzantine consensus without either losing too much detail or becoming impractically complex. We build on recent work on the atomic distributed object model to fill this void with a novel abstraction called AdoB. In addition to revealing important insights into the essence of consensus, this abstraction has practical benefits for easing distributed system verification. As a case study, we proved safety and liveness properties for AdoB in Coq, which are the first such mechanized proofs to handle benign and byzantine consensus in a unified manner. We also demonstrate that AdoB faithfully models real consensus protocols by proving it is refined by standard network-level specifications of Fast Paxos and a variant of Jolteon.

CCS Concepts: • **Software and its engineering** → **Software verification**; **Distributed programming languages**; **Software safety**; **Formal software verification**; **Software reliability**; • **Theory of computation** → **Distributed computing models**; **Abstraction**.

Additional Key Words and Phrases: distributed systems, consensus protocols, byzantine, liveness, formal verification, refinement, proof assistants

ACM Reference Format:
Wolf Honoré, Longfei Qiu, Yoonseung Kim, Ji-Yong Shin, Jieung Kim, and Zhong Shao. 2024. AdoB: Bridging Benign and Byzantine Consensus with Atomic Distributed Objects. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 109 (April 2024), 45 pages. https://doi.org/10.1145/3649826

1 INTRODUCTION

Replication is a powerful tool for systems where data reliability and availability are critical, such as databases or file systems. However, this only works if the replicas agree on the data, which is why consensus protocols, such as Paxos [Lamport 1998] and Raft [Ongaro and Ousterhout 2014], are often

[*]This is the extended technical report, which includes supplementary appendices after the References section.
[†]Wolf Honoré is now at CertiK.

Authors' addresses: Wolf Honoré, Yale University, New Haven, USA, wolf.honore@yale.edu; Longfei Qiu, Yale University, New Haven, USA, longfei.qiu@yale.edu; Yoonseung Kim, Yale University, New Haven, USA, yoonseung.kim@yale.edu; Ji-Yong Shin, Northeastern University, Boston, USA, j.shin@northeastern.edu; Jieung Kim, Inha University, Incheon, South Korea, jieungkim@inha.ac.kr; Zhong Shao, Yale University, New Haven, USA, zhong.shao@yale.edu.

ACM 2475-1421/2024/4-ART109
https://doi.org/10.1145/3649826

at the core of these systems [Burrows 2006; Chang et al. 2006; etcd Authors 2022; Ghemawat et al. 2003]. Unfortunately, these protocols are notoriously complex and easy to implement incorrectly. Formal verification can provide the strongest assurance of their correctness, but this remains a challenging problem because of the inherent complexity of coordinating concurrent, failure-prone servers and an asynchronous network.

The situation becomes even worse when one considers other failure models. Paxos and Raft assume a "benign" setting, such as a data center, where servers are assumed to be cooperative and, at worst, can become unresponsive. However, as the use of consensus in less controlled environments, such as blockchains, becomes more prevalent, so too does the need for formal verification of byzantine consensus protocols [Lamport et al. 1982]. These tolerate a certain number of malicious participants by adding additional rounds of communication to make up for the loss of trust between servers. Though byzantine protocols can tolerate benign failures as well, benign protocols still have their place as they are generally more performant.

Why a new model? In both the benign and byzantine settings, abstraction is the key to scalable verification. The standard approach is to model a protocol as a set of servers with local state that pass messages over an abstract network. Such network-based abstractions are faithful to real system behaviors, but they inherit too many implementation details about network communication, which are largely independent from the essence of the protocol.

Honoré et al. [2022] used a higher-level abstraction called the atomic distributed object (ADO) model to disentangle these concerns and verify the safety of benign consensus extended with a generic hot reconfiguration scheme. This is a promising approach, but it is specific to benign consensus. In fact, nearly all prior verification work considers either just the benign [Hawblitzel et al. 2015; Woos et al. 2016] or just the byzantine [Mazieres 2015; Rahli et al. 2018] setting.

It is not immediately clear that the gap between byzantine and benign protocols can be bridged. The lack of trust between servers seems to demand fundamental changes, and indeed, early implementations, such as PBFT [Castro and Liskov 1999], differ in many ways from their benign predecessors. However, Lamport [2011] identified that the standard benign Paxos can be transformed into a similar byzantine version through refinement, and, in more recent protocols, such as HotStuff [Yin et al. 2019] and Jolteon [Gelashvili et al. 2022], the intuitive structural similarity between the protocols is clearer [Abraham et al. 2021].

Until now, this connection has remained fairly informal, without a clear abstraction to highlight exactly what the key similarities and differences are. In this paper, we present such an abstraction based on the ADO model called AdoB (atomic distributed objects for benign/byzantine consensus). This demonstrates that benign and byzantine consensus use the same basic mechanisms and that, by maintaining a clear separation between network-level communication details and core protocol-level behaviors, one can paper over the superficial differences to obtain a unified model.

Why a unified model? The primary advantage of a single high-level model that captures both benign and byzantine consensus behaviors is that it provides valuable insights into the fundamental nature of consensus and helps to identify and distinguish universal invariants from implementation-specific details. This benefits programming language researchers and system designers alike by clearly separating the concerns of reasoning about the generic class of consensus protocols and proving a particular implementation correct, which leads to simpler and more reusable proofs.

We demonstrate this claim by implementing the AdoB model in the Coq proof assistant [Coq Development Team 2022] and proving that it satisfies both safety and liveness. These are the first proofs to cover both benign and byzantine consensus simultaneously, as well as one of the only mechanized liveness results. Liveness is known to be particularly challenging because one must show that every valid system state eventually transitions to another valid state. In a standard network-based model, this quickly explodes to an overwhelming number of cases due to the many possible message interleavings and failures. For this reason, most prior consensus verification work handles liveness either informally, under strict assumptions, or not at all. AdoB helps to mitigate the complexity by enabling one to prove safety and liveness once and for all in a simpler atomic model that both benign and byzantine protocols can then be proved to refine.

How general is the model? In order to succeed as a useful abstraction, a unified consensus model must accurately reflect real network-level behaviors while also not overfitting to a particular protocol. We show that AdoB meets both of these requirements by proving that network-based specifications of two protocols, a novel variant of the byzantine Jolteon, and a version of benign Fast Paxos [Lamport 2006], both refine the high-level model. Despite significant differences between the protocols, their refinement proofs share a similar structure, and both benefit from the generic AdoB-level safety and liveness properties.

The primary key to AdoB's generality is how it distills the differences between benign and byzantine consensus into a small set of adjustable parameters. For example, quorum sizes are left unspecified, allowing them to be easily instantiated to support a variety of consensus schemes, from a benign f of $2f + 1$ majority to a byzantine proof-of-stake [Saleh 2021] system. In general, nearly any protocol that achieves consensus through gathering quorums of votes over 2–3 rounds should be compatible with AdoB.

Most prior work on verified byzantine consensus does not prove as strong relation between the high-level specification and actual implementations as our refinement, but we found it to be essential for catching bugs in early versions of the model. For example, we discovered subtle errors in our initial attempts to model timeouts in AdoB only after failing to prove refinement.

Our contributions are as follows:

- AdoB: A novel and generic abstraction that unifies benign and byzantine consensus. We also provide an implementation of AdoB in Coq, as well as three instantiations of the parameters for common failure models: benign faults with a simple majority quorum, and byzantine faults with a 2/3 supermajority or a proof-of-stake-style weighted majority.
- Coq proofs of safety and liveness for AdoB, which are the first to handle benign and byzantine consensus in a unified manner.
- This is the first, to our knowledge, mechanized liveness proof for byzantine consensus under a partial synchrony [Dwork et al. 1988] assumption. See Sections 5 and 7 for a comparison with other liveness results. AdoB is also the first variant of the ADO model [Honoré et al. 2021] to support reasoning about liveness at all.
- A novel family of Jolteon variants called GenJolteon, which can be instantiated to tolerate a variety of failure modes.
- Proofs that low-level network-based Coq specifications of GenJolteon and Fast Paxos refine AdoB, thereby benefiting from its safety guarantees.

The Coq and OCaml code that supports these claims is available on Zenodo [Honoré et al. 2024].

2 OVERVIEW

The goal of AdoB is to unify benign and byzantine consensus using the ADO model. Before demonstrating how it achieves this, we briefly review some important background.

2.1 Benign Consensus

Consensus Primer. The goal of consensus is to facilitate agreement across a set of servers (or *replicas*). In particular, we focus on the replicated state machine [Schneider 1990] approach where each replica maintains a log of commands. Replicas may temporarily disagree on certain entries

```
// Leader
elect() {
  time += 1;
  votes := bcast(Elect, time, log);
  return isQuorum(votes); }
local_update() {
  log.append(new_command(time));
  return true; }
commit() {
  votes := bcast(Commit, time, log);
  return isQuorum(votes); }
```

```
// Replicas
handle_elect(m_ldr, m_time, m_log) {
  if (time < m_time)
      && (log.last.time <= m_log.last.time) {
    time := m_time;
    send(m_ldr, ElectAck); } }
handle_commit(m_ldr, m_time, m_log) {
  if (time <= m_time)
     && (log.last.time <= m_log.last.time) {
    time := m_time;  log := m_log;
    send(m_ldr, CommitAck); } }
```

Fig. 1. Benign consensus pseudocode.

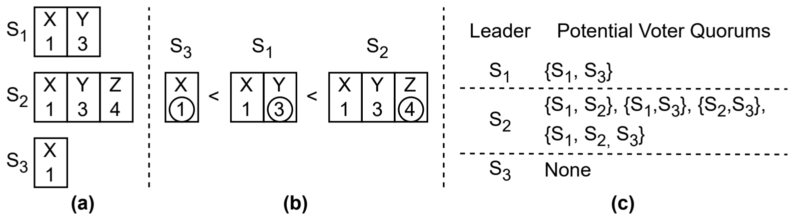


Fig. 2. Deciding which servers can become a leader. **(a)** Servers have a log of timestamped commands. **(b)** Logs are ordered by the timestamp of their last entries. **(c)** A leader may be elected by a quorum of voters with less or equally-recent logs.

towards the tail of the log, but the key safety property is that there always exists a common prefix of *committed* commands on which some *quorum* of replicas agree.

Most consensus protocols, such as Paxos and Raft, accomplish this by repeating three steps: election, local update, and commit (see pseudocode in Fig. 1). The election phase selects a *leader*, which communicates with external clients and coordinates the other replicas for the duration of its term. The precise election mechanism varies by protocol, but it must guarantee that the leader has the most "recent" log among at least a quorum of voters (see Fig. 2). This is decided by comparing by the logical timestamps of the logs' last entries. Once elected, the leader appends a new command to its local log, which is then replicated in the commit phase. If the leader's log is still up-to-date, replicas update their logs to match, and, if a quorum do so, the new command is committed. Note that, in practice, there are many optimizations and fast-paths that can improve performance under normal conditions. Nevertheless, even optimized protocols, at their core, follow this general three-phase template.

Safety and Liveness. The key to maintaining safety through all of this is the fact that elections and commits both require a quorum of voters. Since quorums are defined such that any two quorums have a non-empty intersections (a simple majority is common), this implies that any pair of an election and commit has at least one common voter, which is essential for linearizing them. Replicas only vote for election or commit requests with monotonically increasing timestamps, so the existence of the common voter proves one event must have occurred before the other.

In practice, a safe system is not necessarily useful. Consider, for example, a vacuously safe, trivial protocol that does nothing. Therefore, a liveness property is also necessary, which guarantees that new commands are always committed within some finite time. This is complicated by the fact that

```
// Leader
// NEW: isQuorum -> isSQuorum = super quorum
elect() { ... }
precommit() {
  log.append(new_command(time));
  // NEW: Include votes as evidence of
  // successful election
  votes :=
    bcast(PreCommit, time, log, votes);
  return isSQuorum(votes); }
commit() {
  votes := bcast(Commit, time, log, votes);
  ... }
```

```
// Replicas
handle_elect(m_ldr, m_time, m_log) { ... }
// NEW: Confirm that m_ldr has enough votes, and
// that m_log is safe to commit
handle_precommit(m_ldr, m_time, m_log, m_votes) {
  if (self.time <= m_time)
     && (self.log.last.time <= m_log.last.time)
     && validate(m_votes) {
    self.time := m_time;
    send(m_ldr, PreCommitAck); } }
handle_commit(m_ldr, m_time, m_log, m_votes) {
  // NEW: Confirm that m_ldr did precommit
  if ... && validate(m_votes) { ... } }
```

Fig. 3. Byzantine consensus pseudocode. Common code from the benign case is elided.

replicas may crash (become unresponsive) and network messages may be lost or delayed arbitrarily. In fact, in the general case, liveness is impossible to guarantee [Fischer et al. 1985].

Liveness Assumptions. Despite this impossibility result, all is not lost if we simply introduce a few assumptions that can reasonably be expected to hold in practice. Note that none of the following are necessary for safety.

- There exists at least a *quorum of non-faulty replicas* that never crash. For a typical majority quorum, this means at most f out of $2f + 1$ replicas may crash.
- Instead of total asynchrony, we assume a *partially synchronous* network [Dwork et al. 1988]; i.e., after some unknown point, called the global stabilization time (GST), all messages are delivered to non-faulty replicas within some bounded time.
- There is a *fair rotating leader schedule*; i.e., for every logical timestamp there is exactly one replica that may initiate an election. Here, fairness means there is always a finite number of rounds before some non-faulty replica has a turn.
- Non-faulty replicas follow a *productive strategy*; i.e., they perform operations in a timely manner whenever they are able. For example, a non-faulty leader will attempt to commit new log entries after creating them within some finite time.

The main challenge in proving liveness is showing that the system can reach GST without becoming stuck waiting forever for a non-responsive replica. After that point, the rotating leader assumption ensures that a non-faulty leader will be elected who can commit a command. To avoid blocking forever, replicas maintain local timers that reset after elections and trigger a timeout message on expiration. Upon observing a quorum of timeout messages, a replica knows that no command can ever be committed in the current round (as it would also require a quorum of votes), so it can safely advance to the next round. This ensures a steady progression through rounds that eventually results in a successful commit.

2.2 Byzantine Consensus

Byzantine consensus has the same goal as benign consensus: to allow a collection of replicas to eventually reach agreement on a log of commands. The critical difference is that certain replicas may now behave maliciously, e.g., by ignoring valid requests or lying about local state.

Super Quorums. As with benign consensus, some quorum of replicas is required for both elections and commits (Fig. 3). However, it is no longer sufficient to simply require that quorums overlap, as there is no guarantee the common replica is honest. If the common replica were byzantine, then it could, for example, vote in two elections with the same timestamp, so we cannot trust it to linearize events. Instead, operations require a *super quorum* of votes, which must have at least one honest replica in common with every other super quorum. For example, if f out of $3f + 1$ replicas are byzantine then a super quorum could be any set of $2f + 1$, as at least $f + 1$ must be honest.

Another important implication is that replicas can no longer trust the leader. In particular, they cannot be sure during the commit phase that the leader proposed the same log to everyone. Since no individual can be believed, trust is only possible through a super quorum. Therefore, the step after an election, which is a local operation in the benign case, is now a *pre-commit* phase in which replicas approve a commit, providing the leader can prove it received a super quorum of votes.

Assumptions for Byzantine Replicas. In addition to the assumptions from the benign setting, we must introduce a few more to limit the extent to which byzantine replicas can misbehave.

- Just as a quorum of benign replicas must be non-faulty, a super quorum of replicas in a byzantine setting must be honest at all times. Typically this means less than 1/3 of replicas can be byzantine, though Section 4 will show that this can be generalized. As with faulty replicas, we assume these are fixed in advance, but unknown to honest replicas.
- Byzantine replicas are *computationally bounded* and cannot forge cryptographic signatures. Hence, honest replicas can trust the authenticity of the origin and contents of a message.
- We assume there exists a *gossiping* mechanism. If any honest replica receives a broadcast message, then every honest replica will eventually receive that message. This is necessary only to prove liveness, but not safety. While it is possible to remove this condition, it is common assumption in the byzantine consensus literature [Buchman et al. 2019; Gilad et al. 2017] and doing so increases the complexity of the protocol.

HotStuff and Jolteon. In order to understand some of the design decisions in AdoB, it is helpful to be familiar with the basic workings of the HotStuff and Jolteon byzantine consensus protocols. Note, however, that AdoB is not specific to either of these protocols (see Section 6).

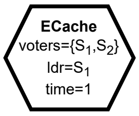
HotStuff and Jolteon follow the usual sequence of phases: election, pre-commit, commit (we consider a two-phase version of HotStuff [Bravo et al. 2020]). In order to overcome the lack of trust between replicas, leaders use *quorum certificates* (QCs) as evidence that an operation is approved (similar to `votes` in Fig. 3). A QC is a collection of a super quorum of cryptographically signed votes [Shoup 2000] containing the identity of the voter, their current timestamp, and the QC for their latest log entry. By collecting a QC with every request, replicas build up a trusted chain of evidence that guarantees byzantine replicas cannot break the safety guarantees.

Once a QC is formed, it is forwarded to the leader for the next round. Under good conditions, the chain of QCs continues to grow; however, a round that ends in a timeout has no QC and breaks the evidence chain. The solution is to fill the gap with a *timeout certificate* (TC). This is similar to a QC, but it contains a super quorum of timeout messages instead of votes, each containing the timed-out replica's latest QC. If a TC is formed, it guarantees no QC can also be formed for the current round, which assures the replicas it is safe to move to the next round.

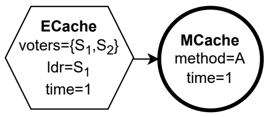
2.3 Atomic Distributed Objects

AdoB uses a modified version of the cache tree abstraction from Honoré et al. [2022]. The key idea is to model not just the current state, but the entire history of a distributed system as a single tree with different nodes (*caches* in ADO terminology) representing the outcome of various operations.

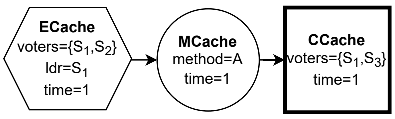
There are three operations for modifying the cache tree: `pull`, `invoke`, and `push` (we omit `reconfig`). Each represents one of the consensus phases (election, local log update, commit), but



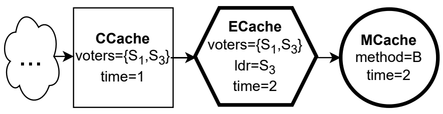
(a) `pull` elects S_1 and adds *ECache*.



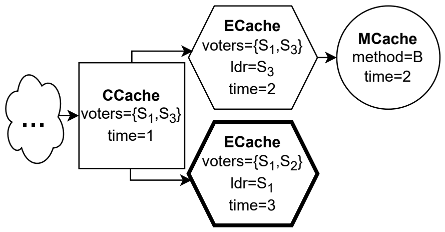
(b) `invoke` adds an *MCache*.



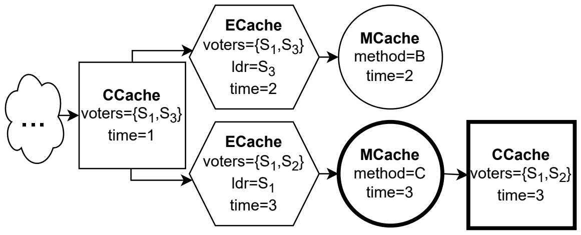
(c) `push` commits and creates a *CCache*.



(d) S_3 is elected and invokes a new method.



(e) S_1 is elected before S_3 commits, creating a fork.



(f) S_1 invokes and commits its own method, making S_3's branch unreachable.

Fig. 4. A cache tree's evolution in the ADO model. Newly created caches are marked with a thick outline. The cloud abbreviates the *ECache*, *MCache* prefix.

the result is decided atomically by consulting a logical oracle rather than through sending network messages. The simplest way to understand these operations is through an example like Fig. 4.

Caches are divided into three variants to represent different operations: *ECache* for elections, *MCache* for method invocations (i.e., local log updates), and *CCache* for commits. Each contains important metadata, such as logical timestamps and quorums of voters. Consider a system consisting of replicas S_1, S_2, and S_3. One must become the leader by calling `pull`, which queries the oracle and indicates that the election either fails or succeeds with some quorum of voters. The `pull` in Fig. 4a receives votes from S_1 and S_2 so it creates an *ECache* for replica S_1. This serves as a logical marker that, at this point, S_1 has the most recent state among at least a quorum of replicas.

Next, S_1 proposes an uncommitted method with `invoke`, which creates an *MCache*. The *MCache* follows the *ECache* to indicate that it is extending S_1's log. The method is then committed using `push`, which again consults an oracle to decide whether a quorum approves it. In this case, both S_1 and S_3 accept the method, so a *CCache* is created, which indicates that the *MCache* is committed.

In the steady state, the tree continues to grow linearly. For example, S_3 may be elected (it voted for the *CCache* so it has the most recent state), after which it can `invoke` another method. Suppose then that S_3 crashes before committing. Eventually, S_1 may become the leader again with votes from S_1 and S_2. Note that neither of these replicas has observed S_3's *MCache* yet. The cache that the *ECache* follows represents the most recent state of its voters, which, in this case, is the *CCache*.

Now there is a fork in the tree, which means there are two competing versions of the state. Fortunately, this inconsistency is resolved as soon as one branch is committed. For example, if S_1

creates a *CCache* with S_1 and S_2, then S_3's branch is effectively unreachable. Any quorum for a later `pull` must contain either S_1 or S_2, so it will choose S_1's *CCache* over S_3's *MCache* because it is more recent. This is the key to guaranteeing the primary safety property that there is a single linear path through the tree containing all *CCaches* (and therefore all committed methods).

One significant advantage of this approach is it abstracts away the details and complexities of network-based communication. Operations either succeed or fail immediately, reducing the number of outcomes to consider. This also provides a uniform, generic interface for consensus that can be implemented by many different protocols. As far as the ADO model is concerned, there is no distinction between a Paxos or Raft election. Any differences are hidden and the common essence is captured by `pull`. Representing the replicas' local states as a tree instead of a set of independent logs also better captures the global dependencies and invariants. For instance, temporary inconsistencies appear as explicit forks in the tree and the committed common prefix can be traced along a branch.

2.4 AdoB

It is clear from Figs. 1 and 3 that benign and byzantine consensus share a similar structure, but there are some key differences, such as the pre-commit phase and the need to validate operations. Rather than attempt to bridge these differences at the implementation level, we instead develop a simplified abstraction (AdoB) for reasoning about high-level properties, and separately prove that it faithfully models these lower-level specifications through refinement. We base AdoB on the ADO model because it has been shown to be effective for high-level reasoning about consensus protocols; however, prior versions are lacking in two areas for our purposes: they have no concept of a timeout, and they are limited to a strictly benign setting.

The first problem is addressed by introducing a new timeout cache (*TCache*) and adjusting `pull`, `invoke`, and `push` to either succeed (creating an *ECache*, *MCache*, or *CCache*, respectively), or fail with a *TCache*. We found this to be a surprisingly subtle operation to model correctly. Recall that timeouts require a set of replicas to communicate amongst themselves without a leader to coordinate them. This is a very different communication pattern than the other operations, and modeling it as an atomic action leads to some surprising behaviors. See Section 6 for a discussion of some subtle bugs we discovered in an early version of AdoB.

By carefully constructing this new timeout-aware ADO model to highlight the essential components of consensus and abstract away any other implementation details, we are able to adapt it to a byzantine setting with only a few additional modifications. The first is, of course, to allow certain replicas to behave maliciously. We model this by relaxing many of the preconditions for `pull`, `invoke`, and `push` to only apply to honest replicas. For example, no restrictions are placed on the local timestamps of byzantine replicas as they cannot be trusted to accurately report them.

The only other significant modification is to change `invoke` from a purely local operation that requires just the leader's approval to one that requires a super quorum of votes. We do this by appealing to an oracle, just as with `pull` and `push`.

The final step is to merge the benign-only and byzantine-only versions of AdoB by observing that the quorum required by `invoke` only needs to be large enough to guarantee a common honest voter with the previous `pull` quorum and following `push` quorum. In the benign setting, the leader is assumed to be honest, so it can serve as the common voter and it is enough for `invoke` to be local, while, in the byzantine case, it requires a super quorum because the leader may be untrustworthy. By introducing a parameterized *method quorum* (*mquorum*), we can cover both cases at once.

3 ADOB FOR BENIGN FAILURES

This section presents a formal specification of the AdoB abstraction specialized to the benign case, along with some key steps of the safety and liveness proofs. Although we do not yet handle

Parameters

$$\begin{array}{rclcrcl} nonfaulty & : & Set(\mathbb{N}_{nid}) & conf \triangleq nonfaulty \cup faulty & isQuorum & : & Set(\mathbb{N}_{nid}) \to \mathbb{B} \\ faulty & : & Set(\mathbb{N}_{nid}) & honest \triangleq conf & leaderAt & : & \mathbb{N}_{time} \to \mathbb{N}_{nid} \end{array}$$

Assumptions

$$(\textsc{Disjoint})\ nonfaulty \cap faulty = \emptyset$$

$$(\textsc{Overlap})\ isQuorum(Q) \land isQuorum(Q') \implies Q \cap Q' \neq \emptyset$$

Fig. 5. Benign AdoB configuration and quorum parameters and assumptions.

$$\begin{array}{rl} Cache \triangleq & ECache(\mathbb{N}_{nid} * \mathbb{N}_{time} * Set(\mathbb{N}_{nid})) \\ & \mid MCache(\mathbb{N}_{nid} * \mathbb{N}_{time} * Set(\mathbb{N}_{nid}) * Method) \\ & \mid CCache(\mathbb{N}_{nid} * \mathbb{N}_{time} * Set(\mathbb{N}_{nid})) \\ & \mid TCache(\mathbb{N}_{time} * Set(\mathbb{N}_{nid}) * Set(\mathbb{N}_{nid})) \end{array} \qquad \begin{array}{rl} CacheTree \triangleq & \mathbb{N}_{cid} \rightharpoonup \mathbb{N}_{cid} * Cache \\ TimeMap \triangleq & \mathbb{N}_{nid} \rightharpoonup \mathbb{N}_{time} \\ \Sigma \triangleq & CacheTree * TimeMap \end{array}$$

Fig. 6. Benign AdoB state definitions.

$$Op \triangleq \texttt{pull} : \mathbb{N}_{nid} \to \Sigma \to \Sigma \mid \texttt{invoke} : \mathbb{N}_{nid} \to Method \to \Sigma \to \Sigma \mid \texttt{push} : \mathbb{N}_{nid} \to \Sigma \to \Sigma$$

Fig. 7. Benign AdoB operations.

byzantine failures, there are several key design decisions that enable a smooth transition to the generalized case in Section 4.

3.1 Semantics

State. Fig. 6 defines the system state (Σ) as a pair of a cache tree and every replica's local logical timestamp (the subscripts on $\mathbb{N}$ are simply labels to clarify the semantic purpose). We use the notations $tree(st)$ and $times(st)$ to discuss these fields. The configuration consists of the disjoint union of an arbitrary set of *nonfaulty* and *faulty* replicas, all of which are *honest* (Fig. 5). The quorum definition is flexible, but it must at least guarantee that any two quorums have a non-empty intersection (Overlap). The rotating leader schedule is determined by the *leaderAt* parameter.

Caches. There are four types of cache representing a successful election (*ECache*), method invocation (*MCache*), commit (*CCache*), or timeout (*TCache*), respectively. Caches are associated with a unique cache ID (*cid*) and the cache tree is implemented as a partial map from a *cid* to its cache and corresponding parent *cid* (with *cid* 0 as the root). New caches can only be added at the leaves of the tree with *addLeaf*, whose definition we omit for brevity.

Each cache contains the logical timestamp (*time*) of the round in which it was created, and the success caches (i.e., not *TCache*) additionally contain the node ID (*nid*) that initiated the operation. Recall that timeouts are initiated independently by several replicas, so *TCaches* instead contain a set of *nids*. Caches are strictly ordered ($\succ$) by comparing timestamps and using *cRank* as a tie-breaker. Fig. 8 defines $\succ$ along with other useful functions on caches and cache trees. We use the variables tr, C, s, and Q to represent cache trees, caches, individual servers, and sets of servers, respectively.

Every cache is associated with two related, but subtly different sets of replicas called its *voters* and *supporters*. A replica's *active* cache (its "local state") is the largest (with respect to $\succ$) for which it is in the set of supporters. Likewise, its voted cache is the largest for which it is in the set of voters. The voter and supporter sets may be equal (as for *CCache*), one may be a subset of the other (*ECache*), or they may be unrelated (*TCache*).

Operations. The AdoB interface consists of `pull`, `invoke`, and `push` (Fig. 7). Each takes its caller's node ID and the current state and returns a new state. The `invoke` operation additionally

$$
\begin{aligned}
cRank(C) &\triangleq \textbf{if } C = ECache(\_) \textbf{ then } 0 \textbf{ else if } C = MCache(\_) \textbf{ then } 1 \textbf{ else} \\
&\quad\ \textbf{if } C = CCache(\_) \textbf{ then } 2 \textbf{ else if } C = TCache(\_) \textbf{ then } 3 \\
C_1 \succ C_2 &\triangleq time(C_1) > time(C_2) \lor (time(C_1) = time(C_2) \land cRank(C_1) > cRank(C_2)) \\
voters(C) &\triangleq \textbf{if } C = ECache(\_, \_, Q) \textbf{ then } Q \textbf{ else if } C = MCache(\_, \_, Q, \_) \textbf{ then } Q \textbf{ else} \\
&\quad\ \textbf{if } C = CCache(\_, \_, Q) \textbf{ then } Q \textbf{ else if } C = TCache(\_, Q, \_) \textbf{ then } Q \\
supporters(C) &\triangleq \textbf{if } C = ECache(nid, \_, \_) \textbf{ then } \{nid\} \textbf{ else if } C = MCache(nid, \_, \_, \_) \textbf{ then } \{nid\} \textbf{ else} \\
&\quad\ \textbf{if } C = CCache(\_, \_, Q) \textbf{ then } Q \textbf{ else if } C = TCache(\_, \_, Q) \textbf{ then } Q \\
voted(tr, s) &\triangleq \max_{\succ} \{C \in tr \mid s \in voters(C)\} \\
active(tr, s) &\triangleq \max_{\succ} \{C \in tr \mid s \in supporters(C)\} \\
activeCommit(tr, s) &\triangleq \max_{\succ} \{C \in tr \mid s \in supporters(C) \land C = CCache(\_)\} \\
canElect(tr, C, Q) &\triangleq (C = CCache(\_) \lor C = TCache(\_)) \land \forall s \in Q \cap honest.\, C \succeq active(tr, s) \\
canInvoke(tr, C, nid, Q) &\triangleq C = ECache(nid, \_, \_) \land \forall s \in Q \cap honest.\, C \succeq voted(tr, s) \\
canCommit(tr, C, nid, Q) &\triangleq C = MCache(nid, \_, \_, \_) \land \forall s \in Q \cap honest.\, C \succeq voted(tr, s) \\
canTimeout(tr, C, Q) &\triangleq \forall s \in Q \cap honest.\, C \succeq activeCommit(tr, s)
\end{aligned}
$$

Fig. 8. Selected benign AdoB auxiliary definitions.

PullOk
$$\frac{\mathbb{O}_{pull}(st, nid) = Ok(Q, C_{max}, t) \qquad st' \triangleq setTimes(st, Q, t) \qquad C_{new} \triangleq ECache(nid, t, Q)}{\mathbb{O} \vdash \texttt{pull}(nid) : st \rightsquigarrow addLeaf(st', C_{max}, C_{new})}$$

InvokeOk
$$\frac{\mathbb{O}_{invoke}(st, nid) = Ok(C_E) \qquad C_{new} \triangleq MCache(nid, time(C_E), \{nid\}, M)}{\mathbb{O} \vdash \texttt{invoke}(nid, M) : st \rightsquigarrow addLeaf(st, C_E, C_{new})}$$

PushOk
$$\frac{\mathbb{O}_{push}(st, nid) = Ok(Q, C_M) \qquad st' \triangleq setTimes(st, Q, time(C_M) + 1) \qquad C_{new} \triangleq CCache(nid, time(C_M), Q)}{\mathbb{O} \vdash \texttt{push}(nid) : st \rightsquigarrow addLeaf(st', C_M, C_{new})}$$

Timeout
$$\frac{\mathbb{O}_{op}(st, nid) = Timeout(Q_{vote}, Q_{supp}, C_{max}, t) \qquad st' \triangleq setTimes(st, Q_{vote} \cup Q_{supp}, t + 1) \qquad C_{new} \triangleq TCache(t, Q_{vote}, Q_{supp})}{\mathbb{O} \vdash op(nid) : st \rightsquigarrow addLeaf(st', C_{max}, C_{new})}$$

Fig. 9. Semantics of benign AdoB operations. Every operation can time out, so Timeout is parameterized by *op*, which can be any of pull, invoke, or push. For invoke, *op* is understood to also take M as an argument.

takes a command to execute on the replicated state machine. As this is completely independent from the safety and liveness properties, we represent it as an abstract, opaque *Method* type.

Network-level failures and asynchrony introduce nondeterminism into the outcome of these operations, which we capture with a logical oracle ($\mathbb{O}$). The oracle abstracts over every way messages may interleave or fail and returns a simple success (*Ok*) or timeout (*Timeout*) result (Fig. 10). The notation $\mathbb{O} \vdash op : st \rightsquigarrow st'$ represents operation op called on state st with oracle $\mathbb{O}$ results in st'.

Pull. The pull operation models an election by asking $\mathbb{O}$ (written as $\mathbb{O}_{pull}$ to indicate the operation under consideration) to choose a set of voters (Q), a sufficiently up-to-date cache (C_{max}), and the next timestamp (t). It then updates the voter's timestamps with *setTimes* to reflect their vote, and adds a new *ECache* child to C_{max} (Fig. 9). This represents a logical marker that at this point, C_{max} is the most recent cache among this quorum of voters.

$$\textsc{ValidPullOracleOk}\quad \frac{\begin{array}{c} t = time(C_{max}) + 1 \qquad leaderAt(t) = nid \qquad isQuorum(Q) \qquad canElect(tree(st), C_{max}, Q) \\ \forall s \in Q \cap honest.\ times(st)[s] \leq t \qquad \forall s \in Q \cap honest.\ time(voted(st, s)) < t \end{array}}{\mathbb{O}_{pull}(st, nid) = Ok(Q, C_{max}, t)}$$

$$\textsc{ValidInvokeOracleOk}\quad \frac{t = time(C_E) \qquad leaderAt(t) = nid \qquad canInvoke(tree(st), C_E, nid, \{nid\})}{\mathbb{O}_{invoke}(st, nid) = Ok(C_E)}$$

$$\textsc{ValidPushOracleOk}\quad \frac{\begin{array}{c} t = time(C_M) \qquad leaderAt(t) = nid \\ isQuorum(Q) \qquad canCommit(tree(st), C_M, nid, Q) \qquad \forall s \in Q \cap honest.\ times(st)[s] \leq t \end{array}}{\mathbb{O}_{push}(st, nid) = Ok(Q, C_M)}$$

$$\textsc{ValidOracleTimeout}\quad \frac{\begin{array}{c} isQuorum(Q_{vote}) \qquad Q_{supp} \cap honest \neq \emptyset \qquad canTimeout(tree(st), C_{max}, Q_{vote}) \\ \forall s \in \left(Q_{vote} \cup Q_{supp}\right) \cap honest.\ times(st)[s] \leq t \qquad \exists s \in Q_{vote} \cap honest.\ times(st)[s] = t \end{array}}{\mathbb{O}_{op}(st, nid) = Timeout(Q_{vote}, Q_{supp}, C_{max}, t)}$$

Fig. 10. Valid benign AdoB oracle conditions. The conditions for timing out are identical regardless of the operation so ValidOracleTimeout is parameterized by *op*.

$\mathbb{O}_{pull}$ chooses these values nondeterministically, but it must obey certain restrictions to faithfully model consensus. The first three are simple sanity checks; namely, the new timestamp follows sequentially from the previous round, the caller is the designated leader for this round, and it has received a quorum of voters. The others ensure the oracle's choice of cache is sufficiently up-to-date. For instance, *canElect* requires that C_{max} is a *CCache* or *TCache*, as those are the only valid ways to end a round, and that it is at least as recent as the honest voters' active caches. The two remaining preconditions guarantee the voters have not already voted for an election with this timestamp.

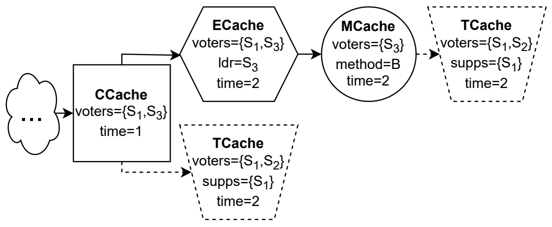
The voters of the new *ECache* are *not* also supporters. They have witnessed the fact that the new leader chose a sufficiently recent cache, but they do not yet have enough evidence to know that setting it as their active cache is safe. For that, they must wait until the leader tells them to commit.

Invoke. The local log update step is modeled by `invoke`. $\mathbb{O}_{invoke}$ simply confirms that it is called by the leader and that the chosen cache (C_E) is that leader's latest *ECache* (*canInvoke*), which it then extends with an *MCache*. This is a local operation that does not require a quorum of approval, so the leader is its sole voter and supporter.

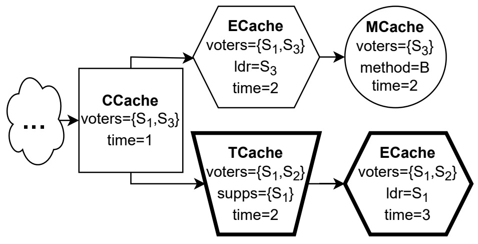
Push. Finally, `push` attempts to commit the *MCache* created by `invoke`. Like `pull` it receives a set of voters (Q), and a cache to commit (C_M) from $\mathbb{O}_{push}$. It performs similar checks to `pull` to confirm the caller is indeed the leader and that C_M is its latest uncommitted *MCache* (*canCommit*). Note that the voters' timestamps are set to one past the *MCache*'s timestamp to ensure that they can no longer participate in the current or any previous rounds.

Now the voters can finally support the *CCache* because the leader has told them it is safe. This influences future `pull` operations because it affects valid choices of C_{max}. Recall that *canElect* requires that C_{max} be at least as recent as its voters' active (i.e., supported) caches. These voters constitute a quorum, which means at least one must also be a supporter of the *CCache*. Therefore, the next election is guaranteed to "see" the *CCache* and choose a C_{max} that is at least as recent.

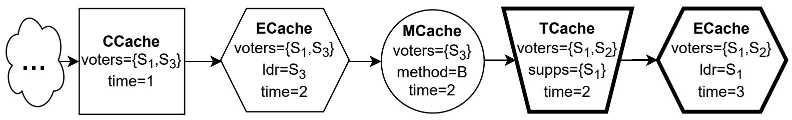
Timeout. For each of these operations, a second possible outcome is a timeout, which is represented by the oracle returning *Timeout* along with the replicas that timed out (Q_{vote}), the replicas



(a) S_3 times out while committing.



(b) Option 1: The next leader starts a new branch.



(c) Option 2: The next leader continues building off the previous *MCache*.

Fig. 11. An example of a timeout in AdoB.

that observed at least a quorum of timeouts (Q_{supp}), the most recent cache among those that timed out (C_{max}), and the timestamp at which they timed out (t). The effect is to create a *TCache*, and, like `push`, force the participating replicas to move to the next round by setting their timestamps to $t + 1$.

The restrictions on the oracle are slightly different from the other cases due to the unique communication pattern used for timeouts. The set of voters, Q_{vote}, have each timed out locally, but it is only when some replicas, Q_{supp}, receive a quorum of these timeout messages that the timeout is considered successful. Therefore, Q_{vote} must be a quorum and Q_{supp} must be non-empty.

Included in each timeout message from Q_{vote} is the replica's active cache. These are collected and forwarded to the leader of the next round to prompt it to begin an election. The oracle enforces this with *canTimeout*, which confirms C_{max} is at least as recent as the voters' latest supported *CCache* (*activeCommit*). The final two preconditions require that no voter or supporter has already timed out or voted in a more recent round, and that at least one voter is actually in the round that is currently timing out. This prevents spurious timeouts for rounds that have not yet even begun.

Though these rules seem reasonable, it is not clear whether some slight modifications might not be equally valid. For example, what if *canTimeout* requires $C = activeCommit(tr, s)$, or Q_{vote} is used for both voters and supporters? These are, in fact, invalid because they do not faithfully model the actual protocol-level behaviors, though this is far from obvious. This demonstrates why refinement is essential to check the validity of the high-level model. Section 6 discusses this further.

Example. As in Fig. 4, in the steady state, branches grow linearly with *ECaches* followed by *MCaches* followed by *CCaches*; however, failures are represented slightly differently with the addition of *TCaches*. Previously, `pull` simply selected the latest *CCache*, which could create forks as in Fig. 4e; now, `pull` must choose a *CCache* or *TCache* from the previous round. This is important to ensure liveness because it prevents `pull` from simply choosing the same *CCache* forever without making any actual progress, but it means the situation in Fig. 4e is now disallowed.

Instead, before creating an *ECache* for time 3, there must first be a *TCache* for time 2. In Fig. 11 the three valid options for the *TCache*'s parent (caches that satisfy *canTimeout*) are: an uncommitted *MCache*, its parent *ECache*, and the latest *CCache*. If the *CCache* is chosen, then a fork is created and the *MCache* is abandoned. Otherwise, if the *MCache* is chosen, then the next leader picks up where the previous one left off and continues extending the same branch. Choosing the *ECache* also creates a fork and is essentially equivalent to choosing the *CCache* because the branch contains exactly the same prefix of *MCaches* and *CCaches*.

3.2 Safety and Liveness Proofs

A practical consensus protocol must be both safe and live. We have proved, in Coq, that both properties hold for AdoB, and, in this section, we summarize some key steps of these proofs as well as some necessary assumptions. Coq versions of the following definitions and theorems can be found in Appendix A and the full proofs can be found in the supplementary materials.

Safety. The top-level safety property is stated as follows.

Theorem 3.1 (Safety). *For any two CCaches in the cache tree, one is a descendant of the other. In other words, committed methods form a linear path through the cache tree.*

The proof proceeds by proving a variety of invariants about well-formed cache trees to show that *CCaches* may never appear on different branches. For example, the following lemma states that every *ECache* must be a descendant of every earlier *CCache*.

Lemma 3.2 (Election Follows Commit). *For any CCache C and ECache C', if $C' > C$, then C' must be a descendant of C.*

This sort of invariant is an example of how the cache tree abstraction can greatly simplify high-level reasoning. Intuitively, it is clear that leaders cannot be elected if they are missing any committed methods. In AdoB it is equally simple to express this formally because *ECaches* and *CCaches* serve as convenient logical markers of when elections and commits occurred relative to each other. A typical network-based model, on the other hand, does not have this level of structure, so formulating this property is much more cumbersome.

This, and several other key invariants, follow from the fact that consecutive elections, timeouts, and commits have overlapping quorums of voters. To keep AdoB as general as possible, we do not specify the exact definition of a quorum, but instead describe it axiomatically by insisting it satisfy the property that two quorums have a non-empty intersection (Overlap in Fig. 5). This permits a range of interesting implementations, some of which are shown in Section 4.2.

Liveness. The liveness of AdoB can be stated informally as: given any cache tree, within some finite time a new method will be committed. To avoid referencing physical time, we formalize this property in terms of a *strategy*.

Definition 3.3 (Strategy). A strategy is a deterministic function that, given a trace of AdoB operations, decides the next operation to execute.

This acts as a logical global scheduler for the replicas, determining what they do and in what order. By repeatedly applying the strategy we can extend the trace and consider future states of the cache tree. For liveness, it is not enough to assume an arbitrary strategy, but instead, we require a *productive* strategy; i.e., one that will try to make progress whenever it is able. This is enforced by requiring that, whenever a replica is able to perform an operation, the strategy will decide to call it within some finite number of steps, and, furthermore, the replica will not participate in any other operations before that point.

Definition 3.4 (Productive Strategy). When a replica is eligible to become the leader, a productive strategy requires it to call `pull` as its next action within a finite number of steps. Similarly, replicas must call `invoke` and `push` as soon as possible whenever they are able.

We can then formally express liveness in the following way.

Theorem 3.5 (Liveness). *Given a cache tree and a productive strategy, within a finite number of steps, a new cache tree will be produced with a more recent* CCache *than the original tree.*

Note that a productive strategy does not require an operation to succeed when called. Due to the partial synchrony assumption, as long as the replica keeps trying it will eventually have an opportunity to succeed. Recall from Section 2.1 that, after some global stabilization time (GST), messages between non-faulty replicas are delivered in finite time, which we express as follows.

Definition 3.6 (Partial Synchrony). There exists an arbitrary but finite GST, as well as a function to determine if a cache tree has reached GST. After GST, if a replica is eligible to be elected, then $\mathbb{O}_{pull}$ returns *Ok* with some set of voters that includes every non-faulty replica. Likewise for $\mathbb{O}_{push}$.

The final necessary assumption is that, a non-faulty leader eventually has the opportunity to be elected. To remain flexible, AdoB simply assumes the existence of an arbitrary deterministic order that eventually selects a non-faulty replica.

Definition 3.7 (Fair Rotating Leadership). Leaders are determined for each round according to some deterministic schedule. The order may be completely arbitrary except that there must be a finite number of rounds between non-faulty replicas.

Armed with these assumptions, the liveness proof decomposes into two main parts: the system always progresses to the next round by either committing a method or timing out; and, after GST, a non-faulty leader is eventually reached. Then, because we have reached GST, Definition 3.6 guarantees the eventual success of `pull` and `push`. The newly created *CCache* must have a strictly larger timestamp than any before it and the proof is complete.

Proof Effort. Implementing benign AdoB in Coq and proving safety and liveness took under one person-month and approximately 700 lines of specification and 6800 lines of proof. This does not include a pre-existing custom library of general lemmas and tactics, nor the initial planning period to design the model and informally outline the proofs. Nevertheless, this is quite fast for mechanized consensus proofs, where timescales are normally on the order of several months rather than weeks. This is largely due to AdoB's atomic interface and cache tree abstraction, which very neatly capture only the essential protocol-level information with none of the orthogonal network-related issues.

4 ADOB FOR GENERALIZED FAILURES

We now demonstrate how to adapt the previous benign model to a byzantine version, and finally merge the two into a generalized abstraction.

4.1 Adapting to Byzantine Consensus

Thanks to our efforts in Section 3 to bring out the shared structure of the benign and byzantine cases, only three additional changes are required to support byzantine consensus. Figs. 12 to 14 highlight these modifications with $\boxed{\text{boxed blue text}}$. The first change is to allow malicious behaviors by partitioning the replicas into *honest* and *byzantine* sets. Now, when preconditions such as *canElect* intersect Q with *honest*, this reflects the fact that byzantine replicas cannot be trusted to accurately report their local state. We still assume that byzantine replicas cannot lie about their identity, invent votes they did not receive, or create caches out of thin air. These are enforced in practice with cryptographic threshold signatures, the implementation of which we do not verify here.

In general, one cannot tell whether an individual replica is honest or byzantine, but, if enough replicas are involved and one assumes an upper bound on the fraction of byzantine replicas, then one can show that the group behaves honestly. This is the purpose of the second change: super quorums (*isSQuorum* in Fig. 12). As with regular quorums, we do not fix super quorums to any

Parameters

$$\boxed{honest} : Set(\mathbb{N}_{nid}) \qquad conf \triangleq \boxed{honest} \cup \boxed{byzantine} \qquad \boxed{isSQuorum} : Set(\mathbb{N}_{nid}) \to \mathbb{B}$$

$$\boxed{byzantine} : Set(\mathbb{N}_{nid}) \qquad isQuorum : Set(\mathbb{N}_{nid}) \to \mathbb{B} \qquad leaderAt : \mathbb{N}_{time} \to \mathbb{N}_{nid}$$

Assumptions

$$(\text{Disjoint}) \; \boxed{honest} \cap \boxed{byzantine} = \emptyset$$

$$(\boxed{\text{SOverlap}}) \; isSQuorum(Q) \land isSQuorum(Q') \implies Q \cap Q' \cap honest \neq \emptyset$$

Fig. 12. Byzantine AdoB configuration and quorum parameters and assumptions. The replicas are no longer all honest. Super quorums must have an honest overlap.

$$\textsc{InvokeOk} \quad \frac{\mathbb{O}_{invoke}(st, nid) = Ok(\boxed{Q}, C_E) \\ \boxed{st' \triangleq setTimes(st, Q \cap honest, time(C_E))} \qquad C_{new} \triangleq MCache(nid, time(C_E), \boxed{Q}, M)}{\mathbb{O} \vdash \texttt{invoke}(nid, M) : st \rightsquigarrow addLeaf(st', C_E, C_{new})}$$

Fig. 13. Semantics of byzantine AdoB operations. All are identical to the benign case except `invoke` now requires a super quorum of voters (Q) instead of just nid.

$$\textsc{ValidInvokeOracleOk} \quad \frac{t = time(C_E) \qquad leaderAt(t) = nid \\ \boxed{isSQuorum(Q)} \qquad canInvoke(tree(st), C_E, nid, \boxed{Q}) \qquad \boxed{\forall s \in Q \cap honest.\ times(st)[s] \le t}}{\mathbb{O}_{invoke}(st, nid) = Ok(\boxed{Q}, C_E)}$$

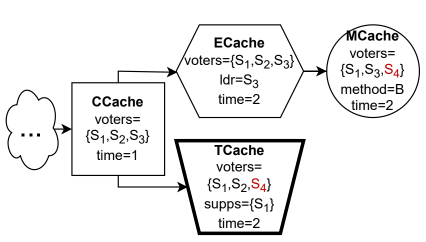
Fig. 14. Valid byzantine AdoB oracle conditions. All cases but `invoke` are identical to Fig. 10 other than replacing *isQuorum* with *isSQuorum*.

particular size, but instead assume only that any two super quorums have a common honest member (SOverlap). Then every instance of *isQuorum* is replaced with *isSQuorum* in Fig. 14.

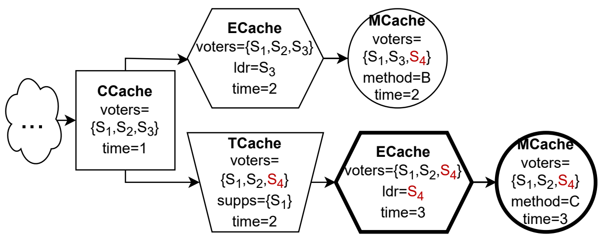
Note that, while the model separates honest and byzantine replicas, it is important that we never rely on this knowledge to determine an operation's outcome. That is why *honest* is only used to weaken preconditions (e.g., $\forall s \in Q \cap honest.\ P(s)$ exempts byzantine replicas from satisfying P). In Section 5, we prove that we do not make any invalid assumptions by showing that they are all satisfiable by a network-level protocol specification.

With these changes, we have moved to a model where only groups, rather than individuals, can be trusted. In particular, this includes the leader, who, if it were byzantine, could attempt to trick other replicas into committing invalid states either by proposing an out-of-date cache, or by equivocating and proposing different caches to different replicas. To rule out this possibility, leaders must gather evidence that at least a super quorum has approved a proposed cache before it can be committed. Previously, this evidence was provided implicitly by `invoke`, with the leader unilaterally giving its approval for an *MCache*. Now, `invoke` must gather a super quorum of voters, which is decided by $\mathbb{O}_{invoke}$ (Fig. 14). The preconditions are the same as before but extended to every replica in Q instead of just the leader. One may wonder if the oracles really capture all possible behaviors of a malicious replica. This is another example of why the refinement proof in Section 5 is critical to validate this high-level model.

Examples. Even with byzantine replicas, AdoB behaves similarly to before. Fig. 15 shows a possible cache tree with one byzantine replica (S_4, shown in red) and three honest replicas (S_1, S_2, S_3). The leader, S_3, successfully invokes a method by acquiring a super quorum of votes (at least 3 out of 4). This ensures that, although one of the voters cannot be trusted (S_4), the other voters form an honest quorum (at least 2 out of 3). At least one of these honest voters must have also voted for the previous election (S_1 and S_3 in this case), so we know creating this *MCache* is safe.

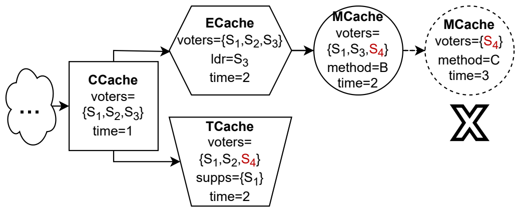


(a) S_1, S_2, S_4 time out waiting for S_3 to commit. S_4 may lie about its time, but it is still safe because $\{S_1, S_2\}$ is an honest quorum.

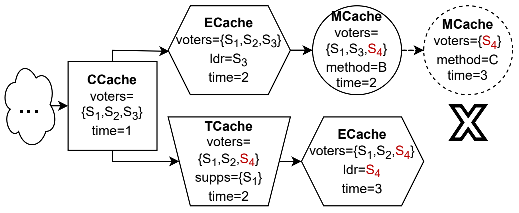


(b) S_4 is elected leader and invokes a method. Byzantine leaders can make progress as long as they behave honestly.

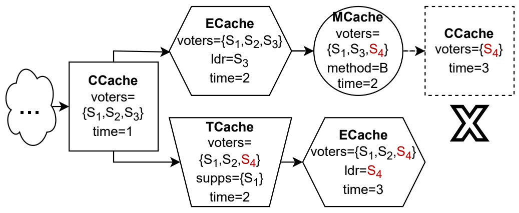
Fig. 15. Allowed behaviors in byzantine AdoB.



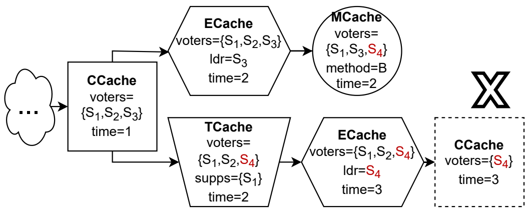
(a) S_4 cannot invoke a method without being elected.



(b) S_4 cannot invoke a method on the wrong branch.



(c) S_4 cannot commit a method from an old round.



(d) S_4 cannot commit without first invoking a method.

Fig. 16. Disallowed behaviors in byzantine AdoB. Dotted outlines represent impossible cases.

In Fig. 15a, S_1, S_2, and S_4 time out while waiting for S_3 to commit and create a *TCache*. It is possible that S_4 is lying about its timer running out, but, once again, the existence of a super quorum of voters ensures the *TCache* is safe despite a potentially malicious participant. Finally, in Fig. 15b, S_4 is successfully elected and invokes a method. This shows that byzantine replicas may sometimes choose to behave honestly, in which case they can contribute to the committed state.

Fig. 16 shows that byzantine replicas are limited in the damage they can cause. For example, S_4 could never create the *MCache* with the dotted outline in Fig. 16a because honest replicas only vote for `invoke` requests from a leader and S_4 does not have an *ECache*. However, even as the leader, S_4 cannot invoke a method on a different branch than its *ECache* because *canInvoke* ensures that the parent of an *MCache* is both an *ECache* and at least as recent as any cache the honest voters have voted for. In Fig. 16b, S_1 and S_2 have voted for the *TCache*, so there is no way to form a super quorum that would vote for S_4's *MCache*.

For the same reasons, S_4 also cannot commit a method from a previous round (Fig. 16c). The *TCache* is more recent than the *MCache* for method B, so S_4 can never acquire enough votes. Nor can it create a *CCache* on its own branch without first invoking a method (Fig. 16d). Replicas require

Parameters

$$\boxed{isMQuorum} : \mathbb{N}_{nid} \to Set(\mathbb{N}_{nid}) \to \mathbb{B}$$

Assumptions

$$(\boxed{\text{MOverlap}})\ isMQuorum(ldr, Q) \wedge isMQuorum(ldr, Q') \implies Q \cap Q' \cap honest \neq \emptyset$$

$$(\boxed{\text{MSOverlap}})\ isMQuorum(ldr, Q) \wedge isSQuorum(Q') \wedge ldr \in Q' \implies Q \cap Q' \cap honest \neq \emptyset$$

Fig. 17. Method quorum (*mquorum*) parameters and assumptions.

ValidInvokeOracleOk

$$\frac{\begin{array}{c} t = time(C_E) \qquad leaderAt(t) = nid \\ \boxed{isMQuorum(nid, Q)} \qquad canInvoke(tree(st), C_E, nid, Q) \qquad \forall s \in Q \cap honest.\ times(st)[s] \leq t \end{array}}{\mathbb{O}_{invoke}(st, nid) = Ok(Q, C_E)}$$

Fig. 18. $\mathbb{O}_{invoke}$ replaces super quorums with *mquorums*.

proof of a successful pre-commit round before voting for a commit request, which in AdoB is modeled by *canCommit*'s requirement that the parent of a *CCache* be an *MCache*.

4.2 Merging the Models

Now, after identifying exactly where these benign and byzantine models differ, we are in a position to unify them by introducing parameters that hide the differences behind a common interface. For two of the changes, this is trivial. The set of byzantine replicas is already a parameter that can simply be instantiated to the empty set for the benign case. Likewise, if *isSQuorum* is set equal to *isQuorum*, then SOverlap clearly holds because quorums overlap and every replica is honest.

This leaves only `invoke`, and the key to bridging this gap is to understand what role `invoke` serves in maintaining an important safety invariant. In order to linearize concurrent events, it is required that, for any two consecutive events, there is a common voter, which creates an unbroken chain of evidence that the logical timestamps are non-decreasing and can therefore be totally ordered. The byzantine case guarantees this by requiring a super quorum of voters for every operation, but, at first glace, the benign case seems to make an exception for `invoke`.

In fact, although benign `invoke` only requires the leader's approval, this does not break the chain of common voters. Observe that an *MCache* always follows an *ECache* created by the same leader, and a *CCache* always follows an *MCache* also from the same leader. Therefore, the leader is the common voter through this chain of caches.

We can therefore consider benign `invoke` to require a special quorum of size 1, whose only restriction is that it must overlap with any other quorum containing the same leader. By dropping the size restriction and generalizing the overlap condition to hold for super quorums, we arrive at a generic *method quorum* (*isMQuorum* in Fig. 17) that can be instantiated to either the benign or byzantine case. Unlike the other quorums, *isMQuorum* depends on the *nid* of the leader as well as a set of voters, which is used to determine when *mquorums* must overlap. In particular, two *mquorums* with the same leader must always have a common honest voter (MOverlap), and an *mquorum* must also have an honest overlap with any super quorum containing the same leader (MSOverlap). All that is needed then to reach the fully unified AdoB model is to replace *isSQuorum* with *isMQuorum* in $\mathbb{O}_{invoke}$'s preconditions (Fig. 18).

Fig. 19 demonstrates that the various quorum parameters can easily be instantiated to support different consensus strategies. In addition to the standard 1/2 benign quorum and 2/3 byzantine super quorums, one can also express something similar to a proof-of-stake scheme [Saleh 2021] in which each replica is assigned a weight ($\mathfrak{w}$), which represents its "voting power". The proofs that these definitions satisfy the overlap assumptions can be found in the supplementary Coq proofs.

	Benign	**Byzantine**	**Weighted (Proof of Stake)**
byzantine	$\emptyset$	Arbitrary $Set(\mathbb{N}_{nid})$	Arbitrary $Set(\mathbb{N}_{nid})$
$isQuorum(Q)$	$\lvert Q\rvert > \lvert conf\rvert/2$	$\lvert Q\rvert > \lvert conf\rvert/2$	$\mathfrak{W}(Q) > \mathfrak{W}(conf)/2$
$isSQuorum(Q)$	$isQuorum(Q)$	$\lvert Q\rvert > 2\lvert conf\rvert/3$	$\mathfrak{W}(Q) > 2\mathfrak{W}(conf)/3$
$isMQuorum(ldr, Q)$	$ldr \in Q$	$isSQuorum(Q)$	$isSQuorum(Q)$

$$\mathfrak{w} : \mathbb{N}_{nid} \to \mathbb{N} \qquad \mathfrak{W}(Q) \triangleq \Sigma_{s\in Q}\mathfrak{w}\,(Q)$$

Fig. 19. Quorum instantiations for benign and byzantine settings.

4.3 Adjusting Safety and Liveness Proofs

Adapting the safety and liveness proofs for benign AdoB to this new unified model is straightforward because all but the essential details have already been stripped away. None of the high-level proof structure changes, and all that remains is to weaken certain lemmas to only apply for honest replicas, and to account for the non-local effects of `invoke`.

Weakening Invariants. AdoB leaves the behavior of byzantine replicas largely unspecified, which means many invariants that previously held for all replicas are now only provable for honest replicas. For example, an honest replica's local time is bounded below by the timestamp of every cache it has voted for or supported, but byzantine replicas can lie about their local time.

As before, everything relies on an honest quorum overlap, this time between super quorums and *mquorums* (SOverlap, MOverlap, MSOverlap). With these additional assumptions, we can show that, even with the weakened invariants, enough honest replicas are involved in every operation that malicious replicas cannot convince the system to behave incorrectly.

***Non-local* `invoke`.** Now that `invoke` requires an *mquorum* of voters, it is no longer a strictly local operation. Therefore, a few new lemmas, as well as some minor changes to existing ones, are required. For example, one important invariant guarantees that `push` appends a *CCache* to the leader's most recent *MCache*.

Lemma 4.1 (Push Max Parent). *If $\mathbb{O}_{push}$ returns Ok for some replica, then the cache it selects is as least as recent (according to $\geq$) as every other MCache created by the same replica.*

In the benign case, this follows from the fact that *canCommit* says C_M is at least as recent as its voters' latest voted caches. Then, when comparing C_M against any other *MCache* C, we know that C's only voter is the leader that created it, which is the same as the current leader by assumption, so $C_M \geq C$. This reasoning does not work in the generalized setting because C now has an *mquorum* of voters. However, because of MSOverlap, we know that C's *mquorum* of voters and `push`'s super quorum of voters have a common honest replica, which means *canCommit* still implies $C_M \geq C$.

Proof Effort. The updated specifications and proofs for the generalized AdoB model required only an additional two person-weeks, approximately 20 lines of specification (720 total), and 1300 lines of proof (8100 total). This relatively small delta is a testament to how well the benign AdoB abstraction already captures the core essence of consensus.

5 SAFETY REFINEMENT AND NETWORK-LEVEL LIVENESS

AdoB's safety and liveness is only meaningful if it faithfully models the behavior of actual benign and byzantine consensus protocols. We demonstrate that this is indeed the case by proving that network-based specifications of two protocols refine AdoB. The first is a novel variant of Jolteon [Gelashvili et al. 2022] that we call GenJolteon because it is capable of tolerating either benign or byzantine faults depending on the instantiation of *mquorum*. The second is Fast Paxos [Lamport

$$\begin{aligned}
\Sigma_{\text{net}} &\triangleq (\mathbb{N}_{nid} \rightharpoonup \mathit{Replica}) * \mathit{Network} \\
\mathit{Replica} &\triangleq \mathbb{N}_{time} * \mathit{Log} * \mathit{Phase} * \mathit{Set}(\mathit{Msg}) \\
\mathit{Network} &\triangleq \mathit{Set}(\mathit{Msg}) * \mathit{Set}(\mathit{Msg}) \\
\mathit{Log} &\triangleq \mathit{List}(\mathbb{N}_{time} * \mathit{Set}(\mathbb{N}_{nid}) * \mathit{Method}) \\
\mathit{Phase} &\triangleq \mathit{NoVote} \mid \mathit{InvokeVoted} \mid \mathit{CommitVoted} \mid \mathit{Done} \\
&\quad \mid \mathit{Elected} \mid \mathit{InvokeWait} \mid \mathit{Invoked} \mid \mathit{CommitWait} \\
\mathit{Msg} &\triangleq \mathit{Request}(\mathbb{N}_{nid} * \mathit{Set}(\mathbb{N}_{nid}) * \mathbb{N}_{time} * \mathit{Cmd}) \\
&\quad \mid \mathit{Ack}(\mathbb{N}_{nid} * \mathbb{N}_{nid} * \mathbb{N}_{time} * \mathit{Cmd}) \\
&\quad \mid \mathit{Timeout}(\mathbb{N}_{nid} * \mathit{Set}(\mathbb{N}_{nid}) * \mathbb{N}_{time} * \mathit{Log})
\end{aligned}$$

$$\begin{aligned}
\mathit{Cmd} &\triangleq \mathit{Elect}(\mathit{Set}(\mathbb{N}_{nid}) * \mathit{Set}(\mathit{Log})) \\
&\quad \mid \mathit{Invoke}(\mathit{Log} * \mathit{Set}(\mathit{Log}) * \mathit{Method}) \\
&\quad \mid \mathit{Commit}(\mathit{Log}) \\
\mathit{Op}_{\text{net}} &\triangleq \texttt{invoke} : \mathbb{N}_{nid} \to \mathit{Method} \to \Sigma_{\text{net}} \to \Sigma_{\text{net}} \\
&\quad \mid \texttt{commit} : \mathbb{N}_{nid} \to \Sigma_{\text{net}} \to \Sigma_{\text{net}} \\
&\quad \mid \texttt{timeout} : \mathit{Set}(\mathbb{N}_{nid}) \to \mathbb{N}_{time} \to \Sigma_{\text{net}} \to \Sigma_{\text{net}} \\
&\quad \mid \texttt{deliver} : \mathit{Msg} \to \Sigma_{\text{net}} \to \Sigma_{\text{net}}
\end{aligned}$$

Fig. 20. Abstract network-based state and operations.

Phase	Leader	Non-leader
NoVote	The replica has entered this round, but has not done anything yet.	
Elected	The leader has received a QC or TC from the previous round and is ready to build an *Invoke* request.	N/A
InvokeWait	The leader has sent out an *Invoke* request and is waiting for responses.	N/A
InvokeVoted	N/A	The replica has voted for an *Invoke* request.
Invoked	The replica has received a super quorum of acks for an *Invoke* request and is ready to send a *Commit* request.	N/A
CommitWait	The replica has sent out a *Commit* request and is waiting for responses.	N/A
CommitVoted	The replica has received a super quorum of *Commit* acks.	The replica has voted for a *Commit* request.
Done	The replica has timed out and will not respond to messages from this round.	

Fig. 21. Semantics of GenJolteon replica phases.

2006], which is a benign protocol with a slightly different voting mechanism from Paxos and PBFT-like protocols. In this section, we give a brief overview of these proofs, as well as a basic performance evaluation for GenJolteon. More technical details can be found in Appendices B and C.

GenJolteon Network-Based Specification. We model the network as a state machine consisting of a set of local replica states and a bag of sent and received messages (Fig. 20). Messages may arrive in any order, at any time after being sent. Honest replicas react by updating their local state and sending new messages. Byzantine replicas are allowed to update their state arbitrarily, but may not do anything that requires forging other replicas' signatures (e.g., constructing a QC). Each replica maintains a local timestamp (the current round it is participating in), a log of methods tagged with a timestamp and a set of voters, a *phase*, and a set of received *Timeout* messages. A replica's phase represents its idea of network progress, and determines what actions it is allowed to take (Fig. 21).

Our notion of refinement consists of proving a relation between network states and cache trees. To reconcile the concurrent, out-of-order network voting events with AdoB's atomic oracular model, we define certain network events as linearization points for cache creation. We then show that every reachable network state has a corresponding valid cache tree, such that there is a bijection between network linearization points and caches. Once this relation is established, we can use AdoB's safety and liveness theorems to prove similar properties for the network-level protocol.

GenJolteon Safety. GenJolteon is based on the standard non-pipelined Jolteon protocol with the same generic quorum parameters as AdoB instead of a fixed 2/3 quorum. GenJolteon uses two phases, invoke and commit, corresponding to the 2-chain rule in Gelashvili et al. [2022]. Each phase requires the leader to collect a super quorum of votes. A successful invoke phase marks a linearization point that corresponds to simultaneously creating an *ECache* and *MCache*. Likewise, a successful commit phase corresponds to creating a *CCache*. By establishing a bijection between these events and AdoB caches, we can prove the following theorem.

Theorem 5.1 (GenJolteon Refinement). *For every valid network state of GenJolteon, there exists a cache tree that is related to the network state through the following refinement guarantees:*

(1) The local log of each replica always corresponds to a branch of the cache tree. If the replica is honest, then the corresponding cache must have a timestamp at least that of the highest CCache the replica voted for;
(2) If the local timestamp of an honest replica is r, then there exists a CCache or TCache of round $r-1$. Hence, the cache tree cannot fall too far behind network progress;
(3) Every successful Commit request (thus, every QC) in the network corresponds to a CCache;
(4) Every MCache in the cache tree corresponds to some proposed block in the network. Therefore, there cannot be spurious blocks in the cache tree.

The first part of the relation, which maps replicas' local logs to cache tree branches, together with AdoB's Theorem 3.1, which says that every *CCache* lies on the same branch, implies GenJolteon's safety property that there is a unique sequence of committed methods that is shared by every replica's log. The proof of this theorem is divided into two major steps. The first involves reordering and grouping related network send and receive events (e.g., votes for the same request), while proving that the resulting honest network state (i.e., all but the byzantine replicas, whose behavior we model non-deterministically) is equivalent to the original order. These events are then collected in a record called the *round descriptor*, which provides a structured view of every externally visible event that has occurred. The second step constructs a cache tree from the round descriptor.

Fast Paxos Safety. The Fast Paxos refinement follows the same network to round descriptor to cache tree approach as GenJolteon; however, aside from only supporting benign failures, there are two differences worth noting. The first is that Fast Paxos is a single-shot protocol that commits at most one value, while AdoB may have arbitrarily many committed *MCaches*. We therefore add the condition to the *canInvoke* predicate that, if the consensus log of the leader's latest *ECache* is not empty, the last entry being m, then the leader may only invoke m again. Then, by induction, the consensus log of every cache is either empty or a repeated sequence of the same method.

The second key difference is that Fast Paxos has two types of rounds: a *slow round*, which works as in standard Paxos where the leader broadcasts a method, and a *fast round*, in which the leader broadcasts a special message that permits voters to accept any method provided by a client directly, bypassing the leader. If clients suggest different methods, the voters may become stuck and time out, which triggers a recovery procedure. We refer readers to Lamport [2006] or Appendix C for details, but a consequence of this voting mechanism is that a 3/4 quorum is necessary.

These different quorum sizes are easily accommodated by AdoB. A super quorum is $3/4$ or more of the voters. For slow rounds, an *mquorum* is just the leader, and, for fast rounds, it is $1/2$ or more of the voters. This implies that any two super quorums intersect on a fast *mquorum*. The linearization point for creating an *ECache* is when a new leader receives a super quorum of timeouts; for an *MCache*, it is when a fast *mquorum* votes for the same value or when the leader decides a value in a slow round; and, for a *CCache*, it is when the leader receives a super quorum of votes.

Compared with GenJolteon, the main verification challenge is showing that the recovery algorithm always returns the committed value, if one exists. Despite the significant differences between the protocols, the overall proof structure is quite similar, primarily involving reordering network events and mapping them to AdoB caches.

GenJolteon Liveness. Unfortunately, whereas GenJolteon's safety follows directly from AdoB's safety, its liveness requires additional network-level reasoning. The problem is the refinement loses important temporal information when it reorders network events. Nevertheless, the safety refinement is still useful for proving the following liveness result. In future work, we plan to investigate alternative forms of refinement that will allow us to use AdoB's liveness more directly.

Theorem 5.2 (GenJolteon Liveness). *After the GST period, starting from any valid network state, a new command will eventually be committed.*

To even state this theorem requires a formal model of time and terms like "eventually". In our liveness proofs, we represent temporal properties in terms of *timed traces.* Let T be the timepoint where GST commences, and Δ be the maximum delivery delay. Then, let τ_k represent the prefix of the timed trace consisting of all events that occurred before timepoint $T + k\Delta$. We can then ask: given the network state at the end of the partial trace τ_k, what can we infer about the network state at the end of τ_{k+1}? For example, consider the scenario where:

- The honest leader of round r is waiting upon a commit request;
- Every honest replica is in round r, and has sent out its commit vote;
- Every honest replica still has at least 2Δ of time at its local timer.

Intuitively, within Δ, the leader will receive all the votes from the honest replicas, and thus its commit request will succeed. We can formalize this idea by considering the network state at $t + \Delta$. First, note that no honest replica could have timed out within Δ, because they all still have sufficient time remaining on their local timers. Therefore, there cannot be a TC of round r at this point.

The rest of the cases follow a similar line of reasoning. For example, if some honest replica has entered a round $r' > r + 1$, then there exists a QC or TC in round $r' - 1$. The structure of the cache tree then implies that there exists a QC or TC in every round between r and $r' - 1$. In particular, this implies the existence of a QC in round r. This demonstrates the main benefit of the refinement with the cache tree model: by referring to the structural properties of the tree, we can infer information about previous events from the current state of the network.

The rest of the liveness proof consists of two parts. First, we show that honest replicas continually enter new rounds. Then, we characterize a set of "good network states" that cover every valid network configuration and prove that each necessarily eventually leads to a successfully committed method. We identify seven such states, supposing that an honest leader is in round r.

(1) Every honest replica is in a round $r' < r$;
(2) Every honest replica is either in a round $r' < r$, or in round r in the *NoVote* phase with timer $\geq 3\Delta$, and at least one honest replica is in round r;
(3) Every honest replica is either in a round $r' < r$, or in round r in the *NoVote* phase with timer $\geq 2\Delta$, or in the *InvokeVoted* phase with timer $\geq 3\Delta$, while the leader is in the *InvokeWait* phase with timer $\geq 3\Delta$;

Layer	Specs (Lines)	Proof (Lines)	Purpose
GenJolteon			
NetworkAtomic	849	4229	Build AdoB cache tree from atomic events.
NetworkMultiElect	801	992	Discard extra TCs.
RoundDescriptor	424	2102	Group individual events into atomic events.
AlmostNetwork	845	6079	Reorder individual events, except timeouts.
NetworkExplicit	753	1298	Reorder receiving timeout messages.
Fast Paxos			
RoundDescriptor	315	1284	Group individual events into atomic events and build AdoB cache tree.
Network	398	813	Reorder individual events.

Table 1. Refinement layers proof effort. See Appendices B and C for descriptions of each layer.

(4) Every honest replica is in round r in the *InvokeVoted* phase with timer $\geq 2\Delta$, while the leader is in the *InvokeWait* phase with timer $\geq 2\Delta$;
(5) Every honest replica is either in a round $r' < r$, or in round r in the *NoVote* phase with timer $\geq \Delta$, or in the *InvokeVoted* phase with timer $\geq \Delta$, or in the *CommitVoted* phase with timer $\geq 3\Delta$, while the leader is in the *CommitWait* phase with timer $\geq 3\Delta$;
(6) Every honest replica is in round r in the *CommitVoted* phase with timer $\geq 2\Delta$, while the leader is in the *CommitWait* phase with timer $\geq 2\Delta$;
(7) The leader is in round r in the *CommitVoted* phase.

If the network is in state 7, then it has received a super quorum of *Commit* acknowledgments. Consequently, from the AdoB safety and refinement proofs, we can conclude that a *CCache* has been created. For any other state, we show that it must progress to another, "better" state with a higher number. For example, suppose that the network is in state 4. Since every honest replica is in the *InvokeVoted* phase, there exists a super quorum of *Invoke* acknowledgments. Since the leader is honest, there is only one *Invoke* request in round r, so everyone acknowledges the same request. After one network step, all of these acknowledgments must have been received by the leader. Therefore, the leader is either in the *CommitWait* or *CommitVoted* phase. In the first case, we reach state 5, and in the second case we reach state 7. See Appendix B.3 for more proof details.

Proof Effort. In total, GenJolteon's refinement and safety proofs took took approximately eight person-months and 17000 lines of Coq proof. Note, however, that this includes the time to discover the right proof structure and correct the GenJolteon and AdoB specifications as errors were discovered. For Fast Paxos, we were able to leverage this experience and common proof architecture to complete the proofs in only one person-month and around 2000 lines of proof. Table 1 summarizes the layers into which each proof was broken. Fast Paxos' proof uses only two layers because we found that GenJolteon's finer-grained steps did not actually reduce the overall proof effort.

GenJolteon's liveness proof took an additional two person-months and 2700 lines of proof. We have not completed a network-level liveness proof for Fast Paxos, but we expect the proof effort to be comparable to GenJolteon's as the informal argument follows essentially the same structure.

(1) Each replica eventually enters a new round due its timer.
(2) After beginning a round, it does not time out within 4Δ.
(3) Once a non-faulty leader enters a round after GST, it can always commit a value within 3Δ.

The primary difference from GenJolteon is that Fast Paxos does not need a pre-commit phase as it does not have to consider byzantine participants. The addition of the fast rounds does not affect the reasoning very much because the proof is mainly concerned with demonstrating progress in the

$$\mathit{canTimeout}(tr, C, Q) \triangleq \forall s \in Q \cap \mathit{honest}.\, C \geq \mathit{activeCommit}(tr, s)$$
$$\wedge \boxed{\exists s \in Q \cap \mathit{honest}.\, C = \mathit{activeCommit}(tr, s)}$$

$$\textsc{ValidOracleTimeout}\quad \frac{\begin{array}{cc} \mathit{isSQuorum}(\boxed{Q}) & \mathit{canTimeout}(\mathit{tree}(st), C_{max}, \boxed{Q}) \\ \forall s \in \boxed{Q} \cap \mathit{honest}.\, \mathit{times}(st)[s] \leq t & \exists s \in \boxed{Q} \cap \mathit{honest}.\, \mathit{times}(st)[s] = t \end{array}}{\mathbb{O}_{op}(st, nid) = \mathit{Timeout}(\boxed{Q}, C_{max}, t)}$$

Fig. 23. An incorrect early attempt at modeling timeouts. The mistakes are marked with a $\boxed{\text{blue box}}$.

worst case, when the recovery procedure is triggered. However, the safety proof already handles much of the complexity by showing that whatever value it produces is safe to commit, and the liveness proof can simply rely on this result.

Extraction to OCaml. To further demonstrate that AdoB faithfully models real protocols, we use Coq's support for extraction to OCaml to produce an executable version of GenJolteon. The pure, functional event handlers are automatically extracted and glued together with a hand-written shim layer that handles network communication. The main execution path of the program is single-threaded and a separate thread manages sending timeout messages as necessary.

We evaluated the extracted code on a research cloud environment with a four-replica configuration. Each node is equipped with four vCPU cores, 16 GB memory, and runs Rocky Linux 8.8. The average network round trip time between nodes is 392 μs. The extracted code exhibits a median latency of 1.87 ms and maximum latency of 9.83 ms (excluding cryptographic signing) to commit a request under a steady state. We configured the timeout to be 10 ms and ran another experiment with one failed replica. Fig. 22 shows a series of latency measurements to increment the timestamp either by committing a method or by timing out. The leader rotates at every timestamp, so the system must wait for a timeout on the failed replica's turn.

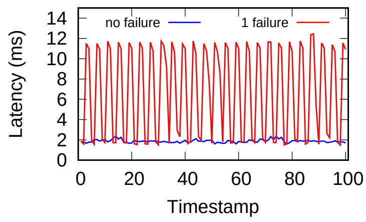


Fig. 22. Latency measurements.

These latency results are comparable to those of the verified instance of PBFT in Rahli et al. [2018] (approximately 1.5 ms), and within an acceptable range of the 0.5 ms achieved by the optimized, unverified BFT-SMaRt system [Bessani et al. 2014]. The extracted code is not optimized for throughput and has a commit rate of 535 blocks per second (a block can include multiple transactions), which is lower than the tens of thousands of transactions per second that BFT-SmaRt and Jolteon [Gelashvili et al. 2022] can achieve. Note that these results are only rough indications of GenJolteon's baseline performance. Our goal is primarily to demonstrate that AdoB can produce executable programs, so there is significant room for relatively simple performance optimizations, including handling requests with multiple threads, batching more transactions per block, and implementing pipelining. In addition to the shim layer, the trusted computing base consists of Coq's extraction mechanism, the OCaml compiler, and the network, thread, and cryptographic libraries.

6 DISCUSSION

Refinement as a Sanity Check. Working at a high level of abstraction is useful for simplifying reasoning, but it can be easy to lose sight of the underlying system. Refinement is an essential tool to sanity check the model against a real implementation and have confidence in its validity. For example, an early version of AdoB had complete safety and liveness proofs, but, during the GenJolteon refinement, we discovered subtle mistakes related to the handling of timeouts (Fig. 23).

One bug was due to incorrectly conflating *TCache* voters and supporters. Recall that a timeout is successful when some replica receives a super quorum of timeout messages. These are bundled together to form a TC, which acts as evidence that it is safe to begin a new round. In AdoB, the TC is represented by a *TCache*, and an oracle determines what super quorum of replicas timed out.

This super quorum is the *TCache*'s voters, but, initially, it was also defined to be its supporters. This implies that the replicas that time out are exactly the same replicas that receive the completed TC, which is not always the case. Suppose replicas S_1 and S_2 time out but only S_3 receives the messages. S_1 and S_2 vote for the TC because they contribute to its creation, but only S_3 supports the TC because it is the only one to actually observe the TC and update its local state accordingly.

This is solved by returning two sets from the oracle: one (Q_{vote}) that represents the replicas that timed out and another (Q_{supp}) that observed the completed *TCache*. Q_{vote} must be a super quorum, but Q_{supp} can be as small as a single honest replica.

A related bug overly restricted the parent cache that the oracle selects for *TCache* (C_{max}). Originally, *canTimeout* required not just that C_{max} was at least as recent as the voters' *activeCommit*, but that it was also equal to one of these *activeCommit*. The reasoning was that some replicas will support this *TCache*, so, to maintain safety, it should only choose a committed cache.

This becomes a problem when considering the situation where a leader invokes a method but times out before committing it (as in Fig. 11). At the network level, the TC may very well contain the uncommitted method, but this incorrect *canTimeout* does not allow a *TCache* to follow an *MCache*. The solution is to drop the requirement that C_{max} be a *CCache*. This is still safe because, as long as it is at least as recent as the latest *CCache*, the linear chain of *CCaches* will not be broken.

AdoB Generality. We have demonstrated that AdoB is generic in the sense that it captures both benign and byzantine consensus. It also supports a variety of consensus strategies, including the typical 1/2 and 2/3 majority quorums, as well as proof-of-stake-style weighted majorities. It would be interesting, in future work, to study proof-of-work systems like Bitcoin [Nakamoto 2008]. Although they exhibit a similar tree structure to other forms of consensus, they typically provide only probabilistic safety guarantees, which poses additional challenges for verification.

From our experience with proving refinement for GenJolteon and Fast Paxos, we expect supporting other common protocols, such as PBFT and Tendermint [Buchman 2016], to be straightforward as they all follow a similar sequence of phases and rely on overlapping quorums to guarantee agreement. For instance, Tendermint has pre-vote and pre-commit phases that are roughly analogous to `invoke` and `push`. Unlike Jolteon, rather than relying on the leader to provide a QC, replicas gather their own evidence of a command's safety by broadcasting their votes. This removes the need for TCs and a pacemaker because the leader is no longer necessary to make progress. Nevertheless, the result is the same from AdoB's perspective: an honest replica may only commit a command for which it has observed a super quorum of votes.

Earlier versions of the ADO model [Honoré et al. 2021, 2022] have already shown that it supports multiple benign protocols, including several Paxos variants and Raft. In almost all respects, AdoB is a strictly more general model, and can therefore be expected to support a superset of these protocols. For example, although AdoB adds *TCaches*, it can still be implemented by a protocol without timeouts, though liveness guarantees may be forfeited. The few restrictions it introduces, such as allowing only a single *MCache* per round and requiring rotating leadership, are necessary for supporting byzantine failures and liveness reasoning and are not very limiting in practice. The former requirement can be worked around by batching multiple commands into a single commit request, and the latter is still quite flexible as it only requires a very weak form of fairness.

Possible Extensions. AdoB is intended to describe the general behavior of leader-based consensus protocols, but there are a number of important optimizations and extensions that, although currently out of scope, would be interesting targets for future work.

	Benign	Byzantine	Safety	Liveness	Executable	Safety Refinement
AdoB	✓	✓	✓	✓	✓	✓
IronFleet [Hawblitzel et al. 2015]	✓	×	✓	✓	✓	✓
Verdi [Wilcox et al. 2015]	✓	×	✓	×	✓	✓
Taube et al. [2018]	✓	×	✓	×	✓	×
Adore [Honoré et al. 2022]	✓	×	✓	×	✓	✓
QTrees [Cirisci et al. 2023]	✓	✓	✓	×	×	✓
Velisarios [Rahli et al. 2018]	×	✓	✓	×	✓	✓
Carr et al. [2022]	×	✓	✓	×	×	×
Padon et al. [2018]	✓	×	✓	✓*	×	×
Losa and Dodds [2020]	×	✓	✓	✓*	×	×
Berkovits et al. [2019]	×	✓	✓	✓*	×	×

Table 2. Comparison between consensus verification projects.
*: The liveness proof does not cover partially-synchronous protocols.

Pipelining, for example, is an optimization implemented by Jolteon and similar protocols that merges the commit phase for the previous round into the pre-commit phase of the current round. However, the danger of a malicious leader still exists, so a command is not actually considered committed until there are two consecutive commits (a 2-chain commit in blockchain terminology). This breaks the simple correspondence between AdoB's `invoke` and `push` operations and the pre-commit and commit phases. A possible solution is to introduce a modified version of AdoB that combines `invoke` and `push` in the same way as two-chain Jolteon. In this version, a *CCache* would not be truly committed until it is directly preceded by a *CCache* from the previous round. One could then prove that the pipelined AdoB refines the three-phase AdoB.

Reconfiguration, the mechanism by which participating replicas can be added and removed, is an important, but subtle operation for practical consensus systems. Honoré et al. [2022] demonstrated that an ADO-based model can support it, but only for a benign setting. Many blockchain protocols, such as Algorand [Gilad et al. 2017], periodically rotate the subset of the participants that are allowed to propose or vote to commit blocks. This could be modeled in AdoB by maintaining an active set of replicas that can be changed either by `pull` or a new operation. The challenge is then to show that a quorum overlap still exists between caches created by different sets of voters.

In practice, consensus is too slow for certain applications, so many real-world systems use it in conjunction with weaker consistency models [Burrows 2006; Dean 2009; Hunt et al. 2010; Li et al. 2012]. It would be interesting to investigate whether an AdoB-like abstraction could be adapted to these weaker models by keeping the cache tree abstraction, but adjusting the behavior of `pull`, `invoke`, and `push`. One might then be able to consider hybrid-consistency systems through some notion of cache tree composition.

7 RELATED WORK

Formal Verification of Consensus. AdoB is the first abstraction to support the simultaneous verification of benign and byzantine consensus, but prior work has studied each case individually. Table 2 compares a selection of these projects along multiple dimensions; namely, does it target benign or byzantine consensus, does it prove both safety and liveness, can it produce executable code, and, if so, is there any formal connection between the code and the high-level abstraction.

Of the selected benign verification frameworks, IronFleet [Hawblitzel et al. 2015] is the only one to prove liveness, using an embedding of TLA [Lamport 1994] in Dafny [Leino 2010]. Safety is proved in an abstract state-machine model, which can be linked with more concrete implementations through refinement. Unlike AdoB, its strengths lie more in facilitating this refinement than providing a generic, reusable abstraction for reasoning about whole classes of protocols.

Verdi [Wilcox et al. 2015] solves a similar problem by providing a mechanism for specifying a distributed system in Coq using a simplified fault-free network-based model and automatically refining it to a more realistic model using verified system transformers. These transformers automatically perform a very similar process to the manual refinement described in Section 5 and it would be interesting future work to attempt to merge these approaches. As with IronFleet, Verdi does not provide a common atomic abstraction for consensus like AdoB, but instead provides developers with tools to reason about individual systems in a more ad-hoc manner.

Another benign safety verification framework is Taube et al. [2018]. It emphasizes decomposing the system into modules and applying decidable logics to check the invariants of these modules.

Adore [Honoré et al. 2022] is the closest in spirit to AdoB and a direct inspiration for our use of the ADO model [Honoré et al. 2021]. It provides a generic cache tree-based abstraction for benign consensus with reconfiguration and a reusable safety proof. Aside from reconfiguration support, which we leave as future work, AdoB is strictly a generalization of Adore. We expect that proving a refinement between a fixed-configuration version of Adore and AdoB would be straightforward.

Quorum Trees [Cirisci et al. 2023] (QTrees) are another consensus abstraction that represent the state of a consensus protocol as a tree of proposed and committed nodes. Its `ADDED` and `COMMITTED` nodes are similar to *MCaches* and *CCaches*, and `GHOST` nodes correspond to *MCaches* that can no longer be selected as the parent of an *ECache*. One difference is that `ADDED` nodes are updated in-place to become `GHOST` or `COMMITTED`, while AdoB's caches are immutable. The authors provide pen-and-paper proofs of the safety of the abstract model and show that a variety of benign and byzantine protocols refine it, but, to our knowledge, these have not been mechanized. QTrees also do not have a means of representing timeouts and are not suitable for liveness reasoning without modifications, which as we found with the ADO model and AdoB, are non-trivial.

Velisarios [Rahli et al. 2018] is the first framework to provide a mechanized safety proof for byzantine consensus. In particular, it showed the safety of PBFT in Coq using a logic-of-events abstraction, which models a system as a collection of traces of logical events with some order enforced by a happens-before relationship. This is similar to the ADO model in that it captures the history of a distributed system as a collection of events with dependencies, but the structure of the cache tree makes the relation to the concrete state (i.e., logs of commands) more explicit. Velisarios does not consider benign consensus or liveness.

Carr et al. [2022] proves the safety of a generalized specification of HotStuff in Agda [Agda Development Team 2022]. The protocol is modeled as an abstract state transition system with parameters for certain implementation details and assumptions that they must satisfy (as we do for *mquorum*). This shares AdoB's goal of capturing the core behaviors of a protocol so proofs of high-level properties can be reused across implementations; however, it is targeted specifically at HotStuff variants, does not cover benign consensus, and lacks liveness and refinement proofs.

Liveness Verification. Our work includes the first mechanized byzantine consensus liveness proof under partial synchrony, but a series of recent research efforts have proved other models of liveness using decidable fragments of temporal logic. Padon et al. [2018] demonstrated that, for certain fully asynchronous or synchronous protocols, liveness guarantees can be converted to safety guarantees. Berkovits et al. [2019] proved liveness for two asynchronous byzantine consensus protocols, but was unable to obtain liveness results for Byzantine Fast Paxos, a partially-synchronous protocol. More recently, Bertrand et al. [2022] verified the liveness of a protocol that is similar in structure to partially-synchronous protocols, but is ultimately still asynchronous.

Among the applications of the liveness-to-safety reduction, Losa and Dodds [2020] are the first to mechanically prove both the safety and liveness of a widely-deployed byzantine protocol, Stellar [Mazieres 2015]. Instead of traditional quorums, Stellar uses federated agreement, in which each replica chooses a set of replicas to trust (a *quorum slice*). The proof uses the Ivy [Padon et al. 2016] Z3-based prover to show the safety and liveness of a first-order logic encoding of the protocol. The validity of this model is then checked against a more standard specification in Isabelle/HOL [Isabelle Development Team 2022] by showing that axioms in the Ivy model hold in Isabelle. However, there is no mechanically-checked connection between the models nor is there any connection to an executable implementation. Also, because Stellar is an open membership consensus protocol, the notion of liveness is weaker than AdoB's. Specifically, the proof does not cover bounded latency of termination under bounded delivery assumptions.

This is not to suggest that these liveness proofs are less valid than AdoB's or that partial synchrony is the "right" model. There are many models of liveness with varying assumptions and guarantees. AdoB's contribution is to demonstrate a simpler way of reasoning about one of the popular ones, which has proved to be challenging for other approaches to handle.

Connecting Benign and Byzantine Consensus. Others have also noticed the similarities between benign and byzantine consensus and attempted to formalize the connection. However, AdoB is the first, to our knowledge, to provide mechanized safety and liveness proofs, as well as a refinement with a concrete implementation.

Lamport [2011] demonstrated that a byzantine version of Paxos (BPCon) refines a modified version of benign Paxos (PCon). In particular, PCon adds a $1c$ message (pre-commit in our terminology) that asserts a particular value is safe to commit. PCon is proved to be safe in TLAPS [Chaudhuri et al. 2008] and is "byzantinized" by proving that BPCon refines it, showing that both implement consensus despite the malicious replicas.

The $1c$ message serves a similar role to AdoB's *mquorum* in that it is a generic method for asserting the validity of a commit with an adjustable burden of proof depending on the trust model. Thanks to the refinement, PCon's safety implies BPCon's safety, but this proof is specialized to this one instance of benign and byzantine protocols. By raising the level of abstraction to the ADO model, AdoB is able to handle a much more general class of protocols. There is an informal argument for the liveness of BPCon, but no mechanized proof.

Another, more general approach by Rütti et al. [2010] aims to provide a generic specification for benign and byzantine consensus. Once again, the key is to parameterize the pre-commit phase (what they refer to as the validation round) to adjust the evidence required from the leader that a command is safe to commit. The authors demonstrate that these parameters can be instantiated for several concrete protocols, including Paxos and PBFT. This is closer to the level of generality provided by AdoB; however, there are no mechanized proofs of safety or liveness for this algorithm. Furthermore, it is specified in terms of a very abstract network-based model with no formal connection to an implementation.

ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers for their helpful feedback. This material is based upon work supported in part by NSF grants 2019285, 1763399, 2313433, and 2118851, and by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-21-C-4018. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

Ittai Abraham, Heidi Howard, and Kartik Nayak. 2021. Benign HotStuff. https://decentralizedthoughts.github.io/2021-04-02-benign-hotstuff/.

Agda Development Team. 2005–2022. What is Agda? https://agda.readthedocs.io/en/latest/getting-started/what-is-agda.html.

Idan Berkovits, Marijana Lazić, Giuliano Losa, Oded Padon, and Sharon Shoham. 2019. Verification of Threshold-Based Distributed Algorithms by Decomposition to Decidable Logics. In *Computer Aided Verification (CAV '19)*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 245–266. https://doi.org/10.1007/978-3-030-25543-5_15

Nathalie Bertrand, Vincent Gramoli, Igor Konnov, Marijana Lazić, Pierre Tholoniat, and Josef Widder. 2022. Holistic Verification of Blockchain Consensus. In *36th International Symposium on Distributed Computing (DISC 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 246)*, Christian Scheideler (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 10:1–10:24. https://doi.org/10.4230/LIPIcs.DISC.2022.10

Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. 2014. State Machine Replication for the Masses with BFT-SMaRt. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '14)*. IEEE Computer Society, Washington, DC, USA, 355–362. https://doi.org/10.1109/DSN.2014.43

Manuel Bravo, Gregory Chockler, and Alexey Gotsman. 2020. Making Byzantine Consensus Live. In *Proc. of the 34th International Symposium on Distributed Computing (DISC '20)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Dagstuhl, Germany. https://doi.org/10.4230/LIPIcs.DISC.2020.23

Manuel Bravo, Gregory Chockler, and Alexey Gotsman. 2022. Liveness and Latency of Byzantine State-Machine Replication. In *Proc. of the 36th International Symposium on Distributed Computing (DISC '22)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Dagstuhl, Germany. https://doi.org/10.4230/LIPIcs.DISC.2022.0

Ethan Buchman. 2016. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains.* Ph. D. Dissertation. University of Guelph.

Ethan Buchman, Jae Kwon, and Zarko Milosevic. 2019. The Latest Gossip on BFT Consensus. arXiv:1807.04938 [cs.DC]

Mike Burrows. 2006. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, USA) *(OSDI '06)*. USENIX Association, Berkeley, CA, USA, 335–350. https://dl.acm.org/doi/10.5555/1298455.1298487

Harold Carr, Christa Jenkins, Mark Moir, Victor Cacciari Miraldo, and Lisandra Silva. 2022. Towards Formal Verification of HotStuff-Based Byzantine Fault Tolerant Consensus in Agda. In *NASA Formal Methods (NFM '22)*, Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez (Eds.). Springer-Verlag, Berlin, Heidelberg, 616–635. https://doi.org/10.1007/978-3-031-06773-0_33

Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation* (New Orleans, LA, USA) *(OSDI '99)*. USENIX Association, Berkeley, CA, USA, 173–186. http://dl.acm.org/citation.cfm?id=296806.296824

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *Proc. of the 7th USENIX Symposium on Operating Systems Design and Implementation* (Seattle, WA, USA) *(OSDI '06)*. ACM, New York, NY, USA, 205–218. https://doi.org/10.1145/1365815.1365816

Kaustuv C Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. 2008. A TLA+ Proof System. In *Workshop on Knowledge Exchange: Automated Provers and Proof Assistants* (Doha, Qatar) *(KEAPPA '08, Vol. 418)*, Renate Schmidt Stephan Schulz, Piotr Rudnicki, Geoff Sutcliffe, Boris Konev (Ed.). CEUR-WS.org, online, 17–37. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-418/paper2.pdf

Berk Cirisci, Constantin Enea, and Suha Orhun Mutluergil. 2023. Quorum Tree Abstractions of Consensus Protocols. In *Proc. of the 32nd European Symposium on Programming (ESOP '23)*, Thomas Wies (Ed.). Springer Nature Switzerland, Cham, 337–362. https://doi.org/10.1007/978-3-031-30044-8_13

Coq Development Team. 1999–2022. The Coq Proof Assistant. http://coq.inria.fr.

Jeff Dean. 2009. Designs, Lessons and Advice from Building Large Distributed Systems. https://research.cs.cornell.edu/ladis2009/talks/dean-keynote-ladis2009.pdf Keynote from ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware.

Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *Journal of the ACM* 35, 2 (April 1988), 288–323. https://doi.org/10.1145/42282.42283

etcd Authors. 2013–2022. etcd. https://etcd.io/

Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM* 32, 2 (April 1985), 374–382. https://doi.org/10.1145/3149.214121

Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. 2022. Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback. In *Proc. of the 26th International Conference on Financial Cryptography and Data Security* (Grenada) *(FC '22)*. Springer-Verlag, Berlin, Heidelberg. https://fc22.ifca.ai/

preproceedings/35.pdf

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *Proc. of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. ACM, New York, NY, USA, 29–43. https://doi.org/10.1145/945445.945450

Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 51–68. https://doi.org/10.1145/3132747.3132757

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proc. of the 25th Symposium on Operating Systems Principles* (Monterey, CA, USA) *(SOSP '15)*. ACM, New York, NY, USA, 1–17. https://doi.org/10.1145/2815400.2815428

Wolf Honoré, Jieung Kim, Ji-Yong Shin, and Zhong Shao. 2021. Much ADO about Failures: A Fault-Aware Model for Compositional Verification of Strongly Consistent Distributed Systems. *Proc. ACM Program. Lang.* 5, OOPSLA (Oct. 2021). https://doi.org/10.1145/3485474

Wolf Honoré, Longfei Qiu, Yoonseung Kim, Ji-Yong Shin, Jieung Kim, and Zhong Shao. 2024. *Artifact For "AdoB: Bridging Benign and Byzantine Consensus with Atomic Distributed Objects"*. Yale University. https://doi.org/10.5281/zenodo.10727569

Wolf Honoré, Ji-Yong Shin, Jieung Kim, and Zhong Shao. 2022. Adore: Atomic Distributed Objects with Certified Reconfiguration. In *Proc. of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI '22)*. ACM, New York, NY, USA, 379–394. https://doi.org/10.1145/3519939.3523444

Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proc. of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA, USA) *(USENIXATC '10)*. USENIX Association, Berkeley, CA, USA, 11. https://doi.org/10.5555/1855840.1855851

Isabelle Development Team. 1986–2022. What is Isabelle? https://isabelle.in.tum.de/overview.html.

Leslie Lamport. 1994. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems* 16, 3 (May 1994), 872–923. https://doi.org/10.1145/177492.177726

Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169. https://doi.org/10.1145/279227.279229

Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19, 2 (2006), 79–103. https://doi.org/10.1007/s00446-006-0005-x

Leslie Lamport. 2011. Byzantizing Paxos by Refinement. In *Proc. of the 25th International Conference on Distributed Computing* (Rome, Italy) *(DISC '11)*. Springer-Verlag, Berlin, Heidelberg, 211–224. https://doi.org/10.5555/2075029.2075058

Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems* (July 1982), 382–401. https://doi.org/10.1145/357172.357176

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proc. of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (Dakar, Senegal) *(LPAR '10)*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer-Verlag, Berlin, Heidelberg, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*. USENIX Association, Hollywood, CA, 265–278. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li

Giuliano Losa and Mike Dodds. 2020. On the Formal Verification of the Stellar Consensus Protocol. In *Proc. of the 2nd Workshop on Formal Methods for Blockchains (FMBC '20, Vol. 84)*, Bruno Bernardo and Diego Marmsoler (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1–9. https://doi.org/10.4230/OASIcs.FMBC.2020.9

David Mazieres. 2015. The Stellar Consensus Protocol: A Federated Model for Internet-Level Consensus. https://www.stellar.org/papers/stellar-consensus-protocol

Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. *Decentralized Business Review* (2008).

Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 305–319. https://dl.acm.org/doi/10.5555/2643634.2643666

Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. 2018. Reducing Liveness to Safety in First-Order Logic. *Proc. ACM Program. Lang.* 2, POPL (Jan. 2018). https://doi.org/10.1145/3158114

Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: Safety Verification by Interactive Generalization. In *Proc. of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*, Chandra Krintz and Emery Berger (Eds.). ACM, New York, NY, USA, 614–630. https://doi.org/10.1145/2908080.2908118

Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Esteves-Verissimo. 2018. Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq. In *Proc. of the 27th European Symposium on Programming (ESOP '18)*, Amal Ahmed (Ed.). Springer-Verlag, Berlin, Heidelberg, 619–650. https://doi.org/10.1007/978-3-319-89884-1_22

Olivier Rütti, Zarko Milosevic, and André Schiper. 2010. Generic Construction of Consensus Algorithms for Benign and Byzantine Faults. In *Proc. of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '14)*. IEEE Computer Society, Washington, DC, USA, 343–352. https://doi.org/10.1109/DSN.2010.5544299

Fahad Saleh. 2021. Blockchain without Waste: Proof-of-Stake. *The Review of Financial Studies* 34, 3 (2021), 1156–1190.

Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)* 22, 4 (1990), 299–319. https://doi.org/10.1145/98163.98167

Victor Shoup. 2000. Practical Threshold Signatures. In *Advances in Cryptology (EUROCRYPT '00)*, Bart Preneel (Ed.). Springer-Verlag, Berlin, Heidelberg, 207–220. https://doi.org/10.1007/3-540-45539-6_15

Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. 2018. Modularity for Decidability: Implementing and Semi-Automatically Verifying Distributed Systems. In *Proc. of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI '18)*. ACM, New York, NY, USA, 662–677. https://doi.org/10.1145/3192366.3192414

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proc. of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 357–368. https://doi.org/10.1145/2737924.2737958

Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. Planning for Change in a Formal Verification of the Raft Consensus Protocol. In *Proc. of the 5th ACM SIGPLAN International Conference on Certified Programs and Proofs* (St. Petersburg, FL, USA) *(CPP '16)*. ACM, New York, NY, USA, 154–165. https://doi.org/10.1145/2854065.2854081

Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proc. of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. ACM, New York, NY, USA, 347–356. https://doi.org/10.1145/3293611.3331591

These appendices contain additional formal and technical details about the proofs described in the paper. Complete Coq formalizations can also be found in the supplementary materials.

A ADOB SAFETY AND LIVENESS DETAILS

This section contains additional details about certain key definitions and theorems for the safety and liveness of AdoB, including Coq formalizations (slightly simplified for the sake of presentation).

Safety. *CCaches* form a linear path in the cache tree. More specifically, given a well-formed cache tree (`ctree_wf` means a tree is created using `pull`, `invoke`, and `push` according to the rules in for a valid oracle), and two distinct *CCaches*, one must be a descendant of the other (`[c ~> c' | ctree]` means `c'` is a descendant of `c` in `ctree`).

```
Theorem safety (ctree: CacheTree) (wf: ctree_wf ctree) :
  forall (c c': Cache),
    c <> c' ->
    In c ctree -> In c' ctree ->
    is_commit c = true -> is_commit c' = true ->
    [c ~> c' | ctree] \/ [c' ~> c | ctree].
```

The proof proceeds by supposing neither cache is the other's descendant in order to derive a contradiction. We then observe that each *CCache* must have a corresponding *ECache* ancestor as well as some nearest common ancestor. By considering the different positions of these caches, we can see that at least one *ECache* is either the nearest common ancestor, or a descendant or ancestor of it. In each case, we use invariants about well-formed cache trees to derive a contradiction.

For example, we can show that *ECaches* have unique timestamps, which means their corresponding *CCaches* must as well. Then, if the more recent of the two *ECaches* is an ancestor of the earlier *CCache*, this contradicts the following lemma, which states that *ECaches* and *TCaches* descend from earlier *CCaches*.

```
Lemma election_follows_commit (ctree: CacheTree) (wf: ctree_wf ctree) :
  forall (c c': Cache),
    is_commit c = true -> (is_election c' || is_timeout c' = true) ->
    c' > c ->
    [c ~> c' | ctree].
```

We represent assumptions about configurations and quorums in the `QuorumParams` typeclass. This introduces three parameters: an abstract configuration (`Config`), a projection function to a set of replicas (`members`), and a function to decide if a given set of replicas is a quorum of a given configuration (`is_quorum`). The quorum definition must satisfy the property that any two quorums share a common replica, and adding more replicas to a quorum still produces a quorum.

```
Class QuorumParams := {
  Config: Type;
  members: Config -> set NID;
  is_quorum: set NID -> Config -> bool;
  quorum_overlap: forall (S S': set NID) (C: Config),
    incl S (members C) -> incl S' (members C) ->
    is_quorum S C = true -> is_quorum S' C = true ->
    exists (s: NID), In s S /\ In s S';
  quorum_subset: forall (S S: set NID) (C: Config),
    incl S S' -> is_quorum S C = true -> is_quorum S' C = true;
  is_squorum: set NID -> Config -> bool;
  is_mquorum: NID -> set NID -> Config -> bool;
  super_honest_subquorum: forall (S: set NID) (C: Config),
    incl S (members C) -> is_squorum S C = true -> is_quorum (S ∩ honest) C = true;
  mquorum_overlap: forall (ldr: NID) (S S': set NID) (C: Config),
    incl S (members C) -> incl S' (members C) ->
    is_mquorum ldr S C = true -> is_mquorum ldr S' C = true ->
    exists (s: NID), In s S /\ In s S' /\ In s honest;
  msquorum_overlap: forall (ldr: NID) (S S': set NID) (C: Config),
    incl S (members C) -> incl S' (members C) -> In ldr S' ->
    is_mquorum ldr S C = true -> is_squorum S' C = true ->
    exists (s: NID), In s S /\ In s S' /\ In s honest;
}.
```

Liveness. Eventually a new *CCache* will be added to the cache tree.

```
Theorem liveness (ctree: CacheTree) (wf: ctree_wf ctree) :
  exists (n: nat), maxCommit (runStrategy n ctree) > maxCommit ctree.
```

A strategy is defined as a typeclass with a `next_move` function that, given a cache tree, decides what operation to apply next (`pull`, `invoke`, or `push`). A strategy can then be run for any number of steps to determine future states of a cache tree.

```
Class Strategy := { next_move: CacheTree -> CacheTree; }.
Fixpoint runStrategy `{Strategy} (n: nat) (ctree: CacheTree) : CacheTree :=
  match n with | 0 => ctree | S n => runStrategy n (next_move ctree) end.
```

A productive strategy guarantees operations are called in a timely manner.

```
Definition productive_pull `{Strategy} (ctree: CacheTree) (nid: NID) :=
  can_pull ctree nid ->
  exists (n: nat),
    runStrategy n ctree = ctree' /\
    next_step ctree' = pull nid ctree' /\
    (forall (n': nat), 0 <= n' <= n -> not_involved nid ctree (runStrategy n' ctree)).
Definition productive_invoke `{Strategy} (ctree: CacheTree) (nid: NID) := ... (* Similar *)
Definition productive_push `{Strategy} (ctree: CacheTree) (nid: NID) := ... (* Similar *)
Definition productive_strategy `{Strategy} := forall (ctree: CacheTree) (nid: NID),
  productive_pull ctree nid /\ productive_invoke ctree nid /\ productive_push ctree nid.
```

A partially synchronous network has some GST, after which messages are guaranteed to be delivered to honest replicas within a fixed time bound. This is modeled by an arbitrary GST parameter, a function to determine whether GST has been reached, and assumptions that after GST all non-faulty replicas will vote for any valid `pull`, `invoke`, or `push` request.

```
Class PSyncParams := {
  GST: nat;
  time_elapsed: CacheTree -> nat;
  gst_pull: forall (ctree: CacheTree) (nid: NID),
    ctree_wf ctree ->
    GST < time_elapsed ctree ->
    In nid nonfaulty ->
    can_pull ctree nid ->
    exists (vote: set NID) (cmax: Cache) (t: Time),
      pull_oracle ctree nid = Ok vote cmax t /\ incl nonfaulty vote;
  gst_invoke ...; (* Similar to gst_pull *)
  gst_push: ...; (* Similar to gst_pull *)
}.
```

Leaders are chosen according to a deterministic scheme that must always eventually select an honest replica.

```
Class LeaderParams := {
  leader_at: Time -> NID;
  leader_at_fair: forall (t: Time), exists (t': Time) (nid: NID),
    t < t' /\ leader_at t' = nid /\ In nid honest;
}.
```

The global time is the timestamp of the most recent *ECache* or *TCache*.

```
Definition global_time (ctree: CacheTree) (t: Time) :=
  exists (c: Cache),
    In c ctree /\
    is_election c || is_timeout c = true /\
    time c = t /\
    (forall (c': Cache), In c' ctree -> time c' <= t).
```

By proving that the global time always eventually increases, we ensure that the system never gets stuck in a particular round.

```
Lemma round_advances (ctree: CacheTree) (wf: ctree_wf ctree) :
  GST < time_elapsed ctree ->
  forall (t t': Time),
    global_time ctree t ->
    t <= t' ->
    exists (n: nat), global_time (runStrategy n ctree) t'.
```

This follows from the productive strategy assumption. Whatever replica is the leader for the current round must eventually call `invoke` and then `push`. If `push` fails and times out, then the round advances and we are done. Otherwise, if it succeeds, then the next leader must eventually call `pull`, which will also advance the global time by creating either an *ECache* on success, or a *TCache* on timeout.

A replica's local timestamp is bounded below by the timestamps of the caches it has voted for or supported.

```
Lemma local_time_lower_bound (ctree: CacheTree) (wf: ctree_wf ctree) :
  forall (nid: NID) (c: Cache),
    In c ctree ->
    In nid honest ->
    voted nid c = true \/ supports nid c = true ->
    time c <= local_time nid.
```

Now we know that, if one waits long enough, a round will begin with a non-faulty leader. Then, because we have reached GST, partial synchrony guarantees the eventual success of `pull` and `push`.

Finally, we prove that the parent cache chosen by $\mathbb{O}_{push}$ is the most recent of the leader's *MCaches*, which implies. that the newly created *CCache* must have a strictly larger timestamp than any before it, and the liveness proof is complete.

```
Lemma push_max_mcache (ctree: CacheTree) (wf: ctree_wf ctree) :
  forall (c cm: Cache) (ldr: NID) (vote: set NID),
    In c ctree ->
    is_method c = true ->
    nid c = ldr ->
    push_oracle ctree ldr = Ok vote cm ctree ->
```

$$\begin{aligned}
\Sigma_{\text{net}} &\triangleq (\mathbb{N}_{nid} \rightharpoonup \mathit{Replica}) * \mathit{Network} \\
\mathit{Replica} &\triangleq \mathbb{N}_{time} * \mathit{Log} * \mathit{Phase} * \mathit{Set}(\mathit{Msg}) \\
\mathit{Network} &\triangleq \mathit{Set}(\mathit{Msg}) * \mathit{Set}(\mathit{Msg}) \\
\mathit{Log} &\triangleq \mathit{List}(\mathbb{N}_{time} * \mathit{Set}(\mathbb{N}_{nid}) * \mathit{Method}) \\
\mathit{Phase} &\triangleq \mathit{NoVote} \mid \mathit{InvokeVoted} \mid \mathit{CommitVoted} \mid \mathit{Done} \\
&\quad \mid \mathit{Elected} \mid \mathit{InvokeWait} \mid \mathit{Invoked} \mid \mathit{CommitWait} \\
\mathit{Msg} &\triangleq \mathit{Request}(\mathbb{N}_{nid} * \mathit{Set}(\mathbb{N}_{nid}) * \mathbb{N}_{time} * \mathit{Cmd}) \\
&\quad \mid \mathit{Ack}(\mathbb{N}_{nid} * \mathbb{N}_{nid} * \mathbb{N}_{time} * \mathit{Cmd}) \\
&\quad \mid \mathit{Timeout}(\mathbb{N}_{nid} * \mathit{Set}(\mathbb{N}_{nid}) * \mathbb{N}_{time} * \mathit{Log})
\end{aligned}
\qquad
\begin{aligned}
\mathit{Cmd} &\triangleq \mathit{Elect}(\mathit{Set}(\mathbb{N}_{nid}) * \mathit{Set}(\mathit{Log})) \\
&\mid \mathit{Invoke}(\mathit{Log} * \mathit{Set}(\mathit{Log}) * \mathit{Method}) \\
&\mid \mathit{Commit}(\mathit{Log}) \\
\mathit{Op}_{\text{net}} &\triangleq \texttt{invoke} : \mathbb{N}_{nid} \to \mathit{Method} \to \Sigma_{\text{net}} \to \Sigma_{\text{net}} \\
&\mid \texttt{commit} : \mathbb{N}_{nid} \to \Sigma_{\text{net}} \to \Sigma_{\text{net}} \\
&\mid \texttt{timeout} : \mathit{Set}(\mathbb{N}_{nid}) \to \mathbb{N}_{time} \to \Sigma_{\text{net}} \to \Sigma_{\text{net}} \\
&\mid \texttt{deliver} : \mathit{Msg} \to \Sigma_{\text{net}} \to \Sigma_{\text{net}}
\end{aligned}$$

Fig. 24. Abstract network-based state and operations.

```
cm ≥ c.
```

B GENJOLTEON DETAILS

In this section, we specify our GenJolteon protocol, and describe our effort to prove safety and liveness of GenJolteon.

The safety proof takes the form of a refinement between network states and AdoB cache trees. The refinement is done in two major steps. In the first step, we construct a *round descriptor* structure that captures all *externally visible* events in the network. In the second step, we construct an AdoB cache tree from the round descriptor that faithfully captures these events. Each step is realized in two or three refinement layers.

The liveness proof first defines a metric called *current network time* (CNT) that measures network progress. We show that increase of CNT is an indication that a new block is committed. Then we show that CNT will eventually increase. This shows that new blocks will eventually be committed.

B.1 Network Specification of GenJolteon

Network-Based Specification. The global state of the abstract network-based model (Fig. 24) consists of local states of each replica and sets of sent and delivered messages. Each replica maintains a local timestamp (the current round it is participating in), a log of methods tagged with a timestamp and a set of voters, a *phase*, and a set of received *Timeout* messages. The phase of a replica indicates its current idea of network progress, and determines what actions it is allowed to take. The semantics of each phase is shown in Fig. 25, and are explained as we present the operation of GenJolteon.

Three kinds of messages exist. They all contain at least a sender, one or more receivers, and a timestamp. A *Request* message additionally contains a *command*, which can be *Elect*, *Invoke*, or *Commit*. *Ack* messages are responses to *Request* messages, and contain the same command as the request it responds to. *Timeout* messages are sent when a replica times-out, and contains the command log stored by the replica when it times-out.

Initialization of GenJolteon. Initially, all replicas are in round 1 and *NoVote* phase. There is nothing for the replicas to do. Consequently, all honest replicas will time out in round 1, broadcasting *Timeout* messages. When an honest replica receives a super quorum of *Timeout* messages, it builds a TC, wraps it in an *Elect* message, and sends it to the leader of the next round.

Normal Operation of GenJolteon. A new round r begins when its leader receives an *Elect* message of timestamp r, which contains a QC or a TC of timestamp $r - 1$. It updates its timestamp,

Phase	Leader	Non-leader
NoVote	The replica has entered this round, but has not done anything yet.	
Elected	The leader has received a QC or TC from the previous round and is ready to build an *Invoke* request.	N/A
InvokeWait	The leader has sent out an *Invoke* request and is waiting for responses.	N/A
InvokeVoted	N/A	The replica has voted for an *Invoke* request.
Invoked	The replica has received a super quorum of acks for an *Invoke* request and is ready to send a *Commit* request.	N/A
CommitWait	The replica has sent out a *Commit* request and is waiting for responses.	N/A
CommitVoted	The replica has received a super quorum of *Commit* acks	The replica has voted for a *Commit* request.
Done	The replica has timed-out and will not respond to messages of this round.	

Fig. 25. Semantics of GenJolteon replica phases.

transitions into *Elected* phase, builds an *Invoke* request, and broadcasts it to all replicas. *Invoke* requests are considered valid only when they come from the leader of round *r* and are backed by a QC or a TC. Invalid requests are silently ignored by honest replicas. Non-leader byzantine replicas cannot build valid requests, nor can byzantine leaders that have not yet received an *Elect* request. After sending out the *Invoke* request, the leader transitions into *InvokeWait* phase.

The replicas, upon receiving the *Invoke* request, enter the new round, send *Invoke* acks to the leader, and transition into *InvokeVoted* phase. Honest replicas respond to at most one *Invoke* request per round. Hence if some *Invoke* request gains a super quorum of acks, no other *Invoke* request of the same round can also gain a super quorum of acks. This prevents byzantine leaders from attempting to commit two different blocks in a single round.

When the leader receives a super quorum of *Invoke* acks, it enters *Invoked* phase, broadcasts a *Commit* request, and updates its own log. After sending the request the leader enters the *CommitWait* phase. When the replicas receive the *Commit* request, they also update their logs, send out *Commit* acks and enter *CommitVoted* phase. When the leader receives a super quorum of *Commit* acks, it enters *CommitVoted* phase, builds a QC and sends it within an *Elect* message to the next leader. Thus, a round completes and a new round begins.

Faulty Operation of GenJolteon. When GenJolteon cannot progress, because the leader is byzantine or the network is asynchronous, each honest replica will eventually time out by broadcasting a *Timeout* message and entering *Done* phase. At this point, they no longer respond to any message in the current round except other Timeout messages. When all honest replicas time out, they will all receive a super quorum of *Timeout* messages, which allows them to build a TC and enter a new round. The TC will be sent in an *Elect* message to the new leader, so it can initiate requests.

```
// Leader
elect(time, logs) {
  // self.time < time || (self.time == time
   && self.phase == NoVote)
  // Received Req(_, _, time, Elect(_, logs))
  self.time := time;
  self.phase := Elected(logs); }
invoke(logs, m) {
  // self.phase = Elected(logs)
  log := max_log(logs);
  cmd := Invoke(log, logs, m);
  self.phase := InvokeWait(cmd, {});
  broadcast_req(self.time, cmd); }
handle_invoke_ack(votes, vote) {
  // self.phase = InvokeWait(cmd, votes)
  // Received Ack(vote, _, _, cmd)
  votes.add(vote);
  if (is_quorum(votes)) {
    self.phase := Invoked;
    self.log.append(m); } }
commit() {
  // self.phase = Invoked
  cmd := Commit(self.log);
  self.phase := CommitWait(cmd, {});
  broadcast_req(self.time, cmd); }
handle_commit_ack(votes, vote) {
  // self.phase = CommitWait(cmd, votes);
  // Received Ack(vote, _, _, cmd)
  votes.add(vote);
  if (is_quorum(votes)) {
    self.phase := CommitVoted;
    cmd := Elect({self.nid}, {self.log});
    ntime := self.time + 1;
    send_req(ldr(ntime), ntime, cmd); } }
```

```
// Replicas
handle_invoke_req(ldr, time, log, logs, m) {
  // Received Request(ldr, _, time,
  //   Invoke(log, logs, m))
  // self.time < time || (self.time = time &&
    self.phase == NoVote)
  self.time := time;
  self.phase := InvokeVoted;
  send_ack(ldr, time, Invoke(log, logs, m));}
handle_commit_req(ldr, time, log) {
  // Received Request(ldr, _, time,
  //   Commit(log))
  // self.time < time || (self.time = time &&
    self.phase == NoVote or InvokeVoted)
  self.time := time;
  self.log := log;
  self.phase := CommitVoted;
  send_ack(ldr, time, Commit(log)); }

// Common
timeout() {
  // self.phase != Elected or Done
  self.phase := Done;
  broadcast_timeout(self.time, self.log); }
handle_timeout(vote, time, log) {
  // Received Timeout(vote, _, time, log)
  // time >= log.last_entry.time
  self.timeouts.add(id, time, log);
  tos := filter_by_time(time, self.timeouts);
  if (is_quorum(tos)) {
    ntime := time + 1;
    self.time := ntime;
    self.phase := NoVote;
    cmd := Elect(ids(tos), logs(tos));
    send_req(ldr(ntime), ntime, cmd); } }
```

Fig. 26. GenJolteon pseudocode.

When Should a Replica Increase its Timestamp? Our protocol is designed along the following principle: if an honest replica increases its timestamp to r, then it is guaranteed that all honest replicas will eventually increase their timestamp to r. Let us see why violating this principle leads to livelessness.

Variation 1: The current leader jumps to the next round after receiving a quorum of *Commit* acks. All other replicas stay in the current round. Suppose the current leader is honest, while the next leader is byzantine. After the current leader jumps to the next round, all byzantine replicas become silent. At this point, it is impossible to convince the remaining honest replicas to jump to the next round. The protocol is stuck.

Variation 2: Every honest replica jumps to the next round upon voting for a *Commit* request. Suppose that the current leader is byzantine. It delays sending out its *Commit* request, so that it reaches a small number of honest replicas, but other honest replicas time out before receiving the request. At this point all byzantine replicas become silent. Now the honest replicas are split into two groups, neither of which forms a quorum. The protocol is stuck.

Variation 3: Every honest replica jumps to the next round upon voting for a *Commit* request, or receiving a *Timeout* message containing a log with timestamp equal to the replica's current timestamp. This is slightly different from variation 2 above. The difference is that, after the honest replicas are split into two groups, those in the higher round can convince those in the lower round to join, by sending out *Timeout* messages. However, this also means that, when the honest replicas eventually enter the higher round, the leader of that round cannot make successful requests, as some of the honest replicas have already timed-out. This implies that a byzantine leader can prevent an adjacent honest leader from making progress. This is an undesirable fairness issue.

One consequence of enforcing this rule is that a leader cannot timeout if it is currently in *Elected* phase, i.e., it has received an Elect message, but has not yet sent out an Invoke request. If it times-out at this point, then it effectively leaves all other replicas behind, which will affect liveness.

Monotonicity and Idempotency in GenJolteon. GenJolteon is designed to be *monotonic* and *idempotent*. This means that:

(1) If an honest replica will not respond to a message now, then it will never respond to that message in the future;
(2) After an honest replica responds to a message, it will never respond to the same message twice.

For example, suppose that an honest replica is in round r and receives an *Invoke* request from round $r' > r$. Since it is always safe to vote for a valid *Invoke* request, the replica sends out an *Invoke* ack, and sets its timestamp to r' and phase to *InvokeVoted*. At this point, it will never vote for the same Invoke request or any *Invoke* request of round r' again.

If a protocol is not formulated in a monotonic and idempotent way, then an honest replica may be currently in state X, where it will refuse to respond to a message M, but later it transitions into state Y, where it will send a response to message M. If a protocol allows such behavior, then when one reasons about its liveness one always needs to consider cases where message M is delivered when the replica is in state X, or in state Y.

This issue may be mitigated by mandating that each honest replica should buffer messages delivered to it to which it cannot respond at the moment. However, in the byzantine setting we then need to consider how to prevent flooding attacks. Byzantine replicas may send out an unbounded number of messages, to which honest replicas cannot decide how to respond. These messages will fill up the buffer of honest replicas, leading to availability issues.

Our protocol avoids these issues, as every request is either immediately responded to or silently dropped. Since there is no request buffer, there cannot be flooding attacks. Later, we will also see how monotonicity and idempotency helps simplifying liveness proof.

B.2 Safety Proof Details

Building the Round Descriptor. In the AdoB model, sending requests and receiving responses are considered a single atomic event. In reality, however, they are distinct and potentially out-of-order. Thus, an *Invoke* request can arrive at a replica long after the leader has received a super quorum of responses and sent out a *Commit* request, due to network asynchrony. Our first step of refinement is to reorder the events, so that responses to a single request become adjacent and can be grouped into atomic events.

Now, what refinement relation can we claim between the traces before and after reordering? Ideally, each replica should have exactly the same state at the end of the trace. In a non-byzantine setting this would be possible, because each replica acts deterministically. However, in a byzantine

```
Record ElectEv := { elect_logs: list (list Entry); elect_is_from_timeout: bool }.
Record InvokeReq := { invoke_logs: list (list Entry); invoke_is_from_timeout: bool;
 invoke_method: Method; }.
Record InvokeEv := { invoke_req: InvokeReq; invoke_recips: set NID; invoke_resps: set NID }.
Record CommitEv := { commit_log: list Entry; commit_recips: set NID; commit_resps: set NID }.
Definition TimeoutEv : Type := NID * list Entry.

Record RoundDesc := {
  round_desc_elect: set ElectEv;
  round_desc_invoke: set InvokeEv;
  round_desc_commit: option CommitEv;
  round_desc_timeout: set TimeoutEv;
  round_desc_timeout_recv: set NID;
  round_desc_tc: set (list TimeoutEv);
}.
```

Fig. 27. Definition of Round Descriptor Structure

setting, this is impossible. Suppose that a byzantine replica sends out a *Commit* request. Later, it forgets this event and sends out an *Invoke* request. After reordering the events, the *Invoke* request would be sent before the *Commit* request. As byzantine replicas behave non-deterministically, there is no guarantee that the replica arrives at the same state in the two traces. Honest replica states are also affected by reordering, though command logs can be preserved.

Because of this, we cannot claim an exact equivalence of network states before and after reordering the events. Instead, we only maintain externally visible events. This means that every message sent/delivered in the unordered trace is still sent/delivered after reordering, and the command log reported by each honest replica is still the same after reordering. These facts are recorded in a structured record called a *round descriptor*.

Refinement Layers between Network Model and Round Descriptor. Our network model is specified in `NetworkExplicit.v`. We first reorder delivery of Timeout messages, so that each replica receives a super quorum of Timeouts atomically. Timeout messages are different from other messages, in that replicas do not respond to them individually, but only when a super quorum of them are received. They have no effect on the local state of honest replicas, other than that they have to buffer them until they form a super quorum. Hence, it is convenient to assume that Timeouts are delivered not individually but in quorums. The model where Timeouts are delivered atomically is `AlmostNetwork.v`, and refinement is proven in `RefineNetExplicit.v`.

We then insert all other send and deliver events into the round descriptor structure, defined in Fig. 27. It contains a list of all Elect messages the leader received, all Invoke and Commit requests the leader sent, responses to each request, and a list of all Timeouts and TCs sent by each replica.

For a round descriptor to be valid, the recorded events must be causally related. For example, each Invoke request must follow from a received Elect message, and every Elect message must follow from a QC or TC from the previous round. These requirements are set down in `RoundDesc.v`, and the refinement proof is in `RefineAlmostNetwork.v`.

Building the Cache Tree. Our next step in refinement is to actually group the adjacent related events into single atomic events. At this point, we encounter a bit of trouble related to timeouts. Suppose that a previous leader sent out a *Commit* request, but it was only received by a small number of honest replicas, before all replicas time out. Now, it may occur that a one replica receives a super quorum of *Timeout* messages containing the proposed block, while another replica build a TC not containing that block. There would be two different TCs in the network, while only one can be represented by a *TCache*.

Ideally, each leader should only receive one *Elect* message, and make one *Invoke* request. Hence, we can choose the one actually used by the next leader to enter the cache tree. In the byzantine setting this breaks down, because a byzantine leader can accept multiple *Elect* messages and make multiple *Invoke* requests. However, it is still the case that at most one *Invoke* request will eventually gain a super quorum of votes. Hence, we choose the TC that will eventually result in a successful *Invoke* request to enter the cache tree.

Refinement Layers between Round Descriptor and Cache Tree. Between the round descriptor and the AdoB cache tree, there are two refinement layers, called `NetworkMultiElect` and `NetworkAtomic`. They share the common feature that sending a request, delivering it to the replicas, and receiving its acks are done in a single atomic step. Also, within each round replicas timeout simultaneously. The difference is that, in `NetworkMultiElect` multiple different TCs can be generated in a timeout event, while in `NetworkAtomic` a single TC is generated, which is chosen from the multiple TCs according to the discussion above. The two layers are defined in `NetworkMultiElect.v` and `NetworkAtomic.v`, and the refinement proofs are in `RefineRoundDesc.v` and `RefineMultiElect.v`.

We then build the cache tree from a `NetworkAtomic` trace in `RefineNetAtomic.v`. The cache tree is correlated to the round descriptors in the following way:

(1) Each successful *Invoke* request corresponds to an *MCache*, and vice versa. Unsuccessful *Invoke* requests are ignored;
(2) Each successful *Commit* request corresponds to a *CCache*, and vice versa. Unsuccessful *Commit* requests are ignored;
(3) Each TC that eventually leads to a successful *Invoke* request corresponds to a *TCache*, and vice versa. Other TCs are ignored.

The various refinement layers are linked together in `RefineLink.v`. The final refinement theorem is `refine_guarantees_bado`.

B.3 Liveness Proof Details

Our Coq proof of liveness is in `Liveness.v`. The final theorem is `eventually_ccache`.

Network Assumptions. We rely on the partial synchrony model. After a certain time called the GST, every message in the network is delivered to each recipient at least once within a period of Δ after it is sent. We will call every period of Δ a *network step*. Recall that our protocol is fully monotonic and idempotent. Suppose that a message M is sent at time x, and one of its recipients n is in a state where it will respond to that message. When the message M arrives at n, either it will respond to M, thus transitioning to a state where it will silently drop M, or it has already been in such a state. We can thus conclude that, for every message M sent before time $x > GST$, and every recipient n of M, after one network step, n will be in a state where it will not respond to M. This simple observation allows us to make useful inferences about the network state after a network step, based on messages sent before that step.

We also need each honest replica not to time out for a sufficiently long period. We assume that, each honest replica has a timer. The timer is reset to at least 3Δ whenever the replica changes its local timestamp or phase. The replica times-out when and only when the timer runs to 0. Hence, if before a network step the countdown at every honest timer is greater than Δ, then no honest replica should time out during the network step.

Finally, we assume that honest replicas perform local events whenever they can. After each network step, no honest replica should be in *Elected* or *Invoked* phase, because they should have sent out *Invoke* and *Commit* requests, transitioning into *InvokeWait* and *CommitWait* phases.

Inferring Network Progress via Safety Refinement. Our proof makes heavy use of the following kind of reasoning. Suppose that the leader of round r is honest, and before a network step, the local timestamp of that leader has not yet reached r. In that case there can be no *Invoke* request, consequently no *CCache* of round r. Suppose that after a network step, we observe that some honest replica increased its timestamp to $r + 1$. By safety refinement, there must be either a *CCache* or a *TCache* of round r. If we additionally know that no honest replica timed-out during the network step, then the cache in question must be a *CCache*. Hence, we know that a new block has been committed.

Measuring Network Progress. Our proof strategy is thus as follows. For every valid network state, we define a value called the *current network time* (CNT) that measures the current progress of the network. It is a global property of the network state, not the timestamp of any individual replica. We first show that after GST, the CNT value will eventually increase. If CNT increases during a network step, and we know that no honest replica timed out during the step, we can infer that a *CCache* has been created. Hence, we only need to focus on network steps where CNT stays constant. Assuming the current CNT is r and the leader of round r is honest, we will show that after the network step, the network state must be in a certain "good state". Furthermore, good states only get "better" after more network steps. Eventually, the network must reach a state where a *CCache* of round r is created.

Our CNT is defined in terms of sent messages. Let r_1 be the maximum r such that an *Invoke* request of timestamp r exists (1 if no such message exists). Let r_2 be the maximum r such that a super quorum of *Timeout* messages of timestamp r exists (0 if no such r exists). Then CNT is defined to be $\max(r_1, r_2 + 1)$. Because sent messages cannot be unsent, one easily sees that CNT must increase monotonically. Also, because *Invoke* requests and *Timeout* messages are broadcast, after one network step, every honest replica should increase its timestamp to at least CNT.

Good Network States. Suppose that before a network step, the CNT is at most r, and it increases (or stays at) r after the network step. In this case, what can we say about the local state of each replica? We can show that the network state must be in one of the following seven "good types":

(1) Every honest replica is in a round $r' < r$;
(2) Every honest replica is either in a round $r' < r$, or in round r with *NoVote* phase and timer $\geq 3\Delta$, and at least one honest replica is in round r;
(3) Every honest replica is either in a round $r' < r$, or in round r with *NoVote* phase and timer $\geq 2\Delta$, or *InvokeVoted* phase and timer $\geq 3\Delta$, while the leader is in *InvokeWait* phase and timer $\geq 3\Delta$;
(4) Every honest replica is in round r with *InvokeVoted* phase and timer $\geq 2\Delta$, while the leader is in *InvokeWait* phase and timer $\geq 2\Delta$;
(5) Every honest replica is either in a round $r' < r$, or in round r with *NoVote* phase and timer $\geq \Delta$, or *InvokeVoted* phase and timer $\geq \Delta$, or *CommitVoted* phase and timer $\geq 3\Delta$, while the leader is in *CommitWait* phase and timer $\geq 3\Delta$;
(6) Every honest replica is in round r with *CommitVoted* phase and timer $\geq 2\Delta$, while the leader is in *CommitWait* phase and timer $\geq 2\Delta$;
(7) The leader is in round r with *CommitVoted* phase.

If the network is in type 7, then it has received a super quorum of *Commit* acks. Consequently, a *CCache* has been created.

We can show that if a network is currently in one of the good types, then after a network step it must be in another good type with higher type number. Our reasoning mostly uses the following pattern. Suppose that currently we are in type 4. Since every honest replica is in *InvokeVoted* phase, there exists a super quorum of *Invoke* acks. Since the leader is honest, there is only one *Invoke* request in round r, so everyone acknowledges the same request. After one network step, all these acks must have been received by the leader. Therefore, the leader is either in *CommitWait* phase or *CommitVoted* phase. In the first case, we reach type 5, and in the second case we reach type 7.

Latency of GenJolteon. Suppose that an honest leader receives an *Elect* message and sends out an *Invoke* request. At this point, the network must be in good state type 3. Since good states get better in each network step, one easily sees that within 4 network steps either we reach type 7, or the network CNT increases. In either case, we can infer that the honest leader has successfully committed a block. This gives us a latency bound of GenJolteon from sending out *Invoke* requests to committing a block.

Subtle Aspects of Liveness Proof. Here we discuss a few subtle aspects we uncovered as we were constructing our liveness proof. They are often overlooked in paper-based liveness proofs.

Suppose that we are after GST, all replicas are in round r with *NoVote* phase and timer $\geq 3\Delta$. The leader of round r has sent out an Invoke request. According to the partial synchrony assumption, all honest replicas will receive the request within Δ. Now, it would seem natural to infer that every honest replica would vote for the request upon receiving it. The liveness proof of Jolteon [Gelashvili et al. 2022] relies precisely on this assumption. However, this is not exactly accurate. Suppose that some byzantine replicas received the request much earlier than some honest replicas, and they voted for the request like any other honest replica. It might occur that the leader had already received a super quorum of votes before the request reached every honest replica. If the leader then sends out a Commit request, and this Commit request reaches honest replicas before the Invoke request, then even honest replicas would not vote for the Invoke request.

To make this kind of "when-they-receive-they-will-vote" argument complete, we must additionally show that, if the replica chooses not to respond to the request when it arrives, then the leader must have already collected a super quorum of votes. In our GenJolteon protocol, if an honest replica does not respond to an Invoke request in round r, then either it has voted for a Commit request in round r, or it has voted for some other request in round $r' > r$. In either case the safety refinement proof shows that there already exists an MCache of round r, indicating the leader has received a super quorum of votes.

Our GenJolteon is based on the principle that a new round can begin only after the previous round has ended. The new leader has to prove this through a QC or TC. On the other hand, PBFT and its variants support so-called *parallel submit*, where a leader can propose a new entry before previous entries have been committed. A liveness proof for PBFT has been claimed in Bravo et al. [2022]. However, we observe that the PBFT protocol described in Bravo et al. [2022] allows unbounded parallel submit, and is therefore subject to the following availability-liveness dilemma.

Since byzantine leaders can propose blocks before previous ones have been committed, it can send out a very large number of block proposals. The honest replicas do not know which ones are valid, and therefore have to simultaneously participate in a large number of byzantine consensus instances. This can easily congest the network and lead to a denial-of-service attack. Real world PBFT implementations usually counter this issue by requiring each honest replica to dynamically maintain a proposal number threshold, and ignore proposals higher than the threshold. We now show that this can lead to liveness issues. Suppose an honest leader sends out a number of proposals. Since byzantine replicas can make votes like any other honest replica, these proposals may get committed before reaching all honest replicas. Suppose that the leader now makes a proposal k, and

```
Record NodeState : Type := {
  node_round : nat; (* Current round of replica *)
  node_leader_phase : LeaderPhase; (* Progress indicator for leader *)
  node_voter_phase : VoterPhase; (* Progress indicator for voter *)
  node_commit_round : nat; (* Last round in which the voter made votes; 0 if never voted *)
  node_commit_method: nat; (* The value voted for in the last vote *)
  node_commit_fast: bool; (* Whether the last vote was made after an "Any" message *)
  node_recv_timeouts : list TimeoutMsg; (* List of received timeouts *)
  node_recv_votes : ZMap.t (option VoteMsg); (* List of received votes *)
}.
```

Fig. 28. State variables of replicas in Fast Paxos.

k is above the threshold of some honest replica n, who has not yet learned the previous proposals. If proposal k reaches n before the previous proposals, then k gets dropped, even though it is from an honest leader. Hence, naive thresholding is problematic in a formal analysis of liveness. Bravo et al. [2022] simply assumes that a replica should buffer all received messages, but this leads to flooding attacks as we have noted above.

C FAST PAXOS DETAILS

In this section we specify our version of Fast Paxos and describe related proofs.

C.1 Network Specification of Fast Paxos

We first describe our specification of Fast Paxos. The overall framework is similar to GenJolteon. We start from defining replica states and messages, and then define the actions in pseudocode.

Fast Paxos involves more bookkeeping than GenJolteon, because it has to support the conflict recovery mechanism. The replica state is shown in Fig. 28, where we annotate each field to indicate its content. The message schema is shown in Fig. 29. The timeout message, being more complex than other message types, is described in Fig. 30.

In the classic presentation of Fast Paxos, a replica can request to become the leader by sending Phase 1a messages and collect Phase 1b votes. It is difficult to achieve deterministic liveness under this design. Therefore, we replaced it with a pacemaker-like approach where Timeout messages play the role of Phase 1b messages. A replica becomes a leader only after receiving timeouts from 3/4 of voters. The request and vote messages here correspond to Phase 2a and Phase 2b messages in the classic presentation.

The leader phase of a replica can be one of *NotLeader*, *Pulled*, *Requested*, or *Committed*. The voter phase can be one of *NoVote*, *VoteAny*, *Voted*, or *Done*. The semantics are shown in Fig. 31. The pseudocode of how the replica transitions between these phases is shown in Fig. 32. The recovery algorithm, which recovers a potentially committed value from a super-quorum of timeouts, is described in Algorithm 1. The full formal details are in the file `refine/specs/FastPxNetwork.v` file of the artifact.

C.2 Details of Safety Refinement Proof

Like the safety proof for GenJolteon, the Fast Paxos proof proceeds in two steps. In the first step we construct a round descriptor structure that captures key network events and presents a structured view of them. This step is completed in `refine/proofs/RefineFastPxNetwork.v`. The round descriptors provide a convenient language for proving simple invariants about the system, such as

$$
\begin{aligned}
\Sigma_{\text{net}} &\triangleq (\mathbb{N}_{nid} \rightharpoonup NodeState) * Set(Msg) \\
Msg &\triangleq Request(\mathbb{N}_{round} * Method) \\
&\quad \mid RequestAny(\mathbb{N}_{round}) \\
&\quad \mid Vote(\mathbb{N}_{nid} * \mathbb{N}_{round} * Method) \\
&\quad \mid Timeout \\
&\quad \mid TC(\mathbb{N}_{round} * Option(Method))
\end{aligned}
$$

Fig. 29. Network state of Fast Paxos.

```
struct timeout_msg {
  int id; // Voter ID
  int r; // Current round number upon timeout
  int last_commit_round; // Last round in which the voter made commit votes
  int last_commit_value; // The value the voter last voted for
  bool last_commit_is_fast; // Whether the vote was made after "Any" message
}
```

Fig. 30. Timeout message structure in our Fast Paxos

Leader Phase	Meaning	Voter Phase	Meaning
NotLeader	The replica is not leader in this round	*NoVote*	The replica has not voted yet
Pulled	The leader has received TC and is ready to send request	*VoteAny*	The voter has received VoteAny and can choose a value
Requested	The leader has sent out request message	*Voted*	The voter has voted in this round
Committed	The leader has committed a value in this round	*Done*	The voter has timed-out in this round

Fig. 31. Semantics of replica phases in Fast Paxos.

Algorithm 1 Fast Paxos Recovery Algorithm

```
1: function RECOVER_VALUE(timeouts)
2:     $t \leftarrow \arg\max_{msg \in timeouts} msg$.last_commit_round
3:     if $t$.last_commit_round = 0 then return $\perp$
4:     if $t$.last_commit_is_fast = $false$ then
5:         return $t$.last_commit_value
6:     else
7:         if 1/2 of voters commit some single value $m$ in round $t$.last_commit_round then
8:             return $m$
9:         else
10:            return $\perp$
```

```
// Leader
handle_tc(round, method) {
  // self.round < round
  self.round := round;
  if self.id = leader_at(round) {
    self.leader_phase := Pulled(method);
  } else {
    self.leader_phase := NotLeader;
  } }
request(m) {
  // self.leader_phase = Pulled None ||
 self.leader_phase = Pulled (Some m)
  self.leader_phase := Requested
  self.recv_votes := [];
  broadcast(Request(self.round, m)); }
request_fast() {
  // self.leader_phase = Pulled None
  self.leader_phase := Requested
  self.recv_votes := [];
  broadcast(RequestAny(self.round)); }
commit() {
  // self.leader_phase = Requested
  // Received votes for m from 3/4 of
 voters
  self.leader_phase := Committed; }
```

```
// Voter
handle_tc(round, method) {
  // self.round < round
  self.round := round;
  self.voter_phase := NoVote; }
handle_request(round, method) {
  // self.round < round || self.round =
 round && self.voter_phase = NoVote
  self.voter_phase := Voted;
  self.commit_round := round;
  self.commit_value := method;
  self.commit_is_fast := false; }
handle_request_fast(round) {
  // self.round < round || self.round =
 round && self.voter_phase = NoVote
  // Voter chooses method m
  self.voter_phase := Voted;
  self.commit_round := round;
  self.commit_value := m;
  self.commit_is_fast := true; }

//Common
timeout() {
  self.voter_phase := Done;
  broadcast(Timeout(id, self.round, self.
 commit_round, self.commit_value, self.
 commit_is_fast)); }
build_tc() {
  // Received timeouts of round self.round
 from 3/4 of voters
  v := recover_value(timeouts);
  broadcast(TC(self.round, v)); }
```

Fig. 32. Pseudocode of Fast Paxos.

that if a voter sends out both commit votes and timeouts within a round, then the timeout would reflect the commit vote. See `refine/specs/FastPxRoundDesc.v` for more invariants.

In the second step, an AdoB cache tree is constructed from the descriptors. This step is completed in `refine/proofs/RefineFastPx.v`.

Unlike GenJolteon, where it is straightforward to establish the preconditions for add caches, a key part of Fast Paxos safety proof is to prove that the recovery algorithm will always recover a committed value. Our safety reasoning can be summarized as follows. Suppose a value m has been committed in round r, and the leader of round $r' > r$ has received a super-quorum of timeouts from round $r' - 1$. Then at least $1/2$ of voters have both committed value m and sent out timeouts. Suppose that none of these voters have made commit votes in any round $r'' > r$, then the recovery algorithm would observe commit votes for value m from all these voters and return m (Line 8 of Algorithm 1). If some of these voters have made commit votes in some round $r'' > r$, since these votes must have been made after receiving a request message from the leader of round r'', there must exist an *ECache* of round r''. By safety property of AdoB cache tree, that leader must have observed that value m had been committed, and so can only rebroadcast m again. Hence the recovery algorithm would still return m as expected (Line 5 of Algorithm 1).

Received 20-OCT-2023; accepted 2024-02-24