Abstract

The Atomic Distributed Object Model for Distributed System Verification

Wolf Honoré

2022

Distributed systems are at the heart of most web-based applications and are responsible for replicating and maintaining critical data. Unfortunately, their inherent concurrency combined with an asynchronous and unreliable network makes them prone to implementation bugs that can have serious real-world consequences. Formal verification can offer strong assurances of correctness; however, despite recent advances, reasoning directly about a system's implementation remains prohibitively complex. The key is to find the right abstraction that faithfully models a system's behaviors, while avoiding irrelevant implementation details.

This dissertation presents such an abstraction called the atomic distributed object (ADO) model, which hides the existence of the network and reduces all behaviors to three atomic operations. This not only makes the ADO model simpler, which enables more scalable verification, but it also means it is general enough to capture a wide variety of systems. We describe three verification frameworks built around the ADO model, each implemented in the Coq proof assistant and targeted at different problems. The first, Advert, supports compositional reasoning about distributed objects, which can be combined to build more complex applications. The second, Adore, proves the safety of a general class of reconfiguration schemes, which is an essential, but often overlooked, operation for practical distributed systems. Finally, AdoB shows that the ADO model can be used for liveness reasoning, and can express both benign and byzantine failure models in a unified manner.

The Atomic Distributed Object Model for Distributed System Verification

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Wolf Honoré

Dissertation Director: Zhong Shao

December 2022

© 2022 by Wolf Honoré
All rights reserved.

Contents

List of Figures

List of Tables

Acknowledgments

I am deeply grateful to my advisor, Prof. Zhong Shao, for his invaluable guidance and constant support. His passion for research and his dedication to furthering his students' development as researchers are inspiring. I am also thankful to Profs. Ruzica Piskac, James Aspnes, and Benjamin Pierce for graciously agreeing to serve on my dissertation committee.

I would also like to thank the members of the Yale FLINT group for their feedback, camaraderie, and many fruitful discussions. Special thanks go to my frequent collaborators, Jieung Kim and Ji-Yong Shin, without whom this work would not have been possible. It is largely by their example that I learned to write effective research papers. I am also grateful to Longfei Qiu and Yoonseung Kim, who, in our short time working together, have already greatly improved this work through their feedback and assistance.

Finally, my appreciation goes out to my friends and family for their faith, encouragement, and guidance, and especially to my wife, Becky, for her love, support, and patience. Thank you for taking care of Real Life and making the last six years possible.

Chapter 1

Introduction

Distributed systems are programs that are cooperatively executed simultaneously by multiple processes, potentially on different, physically distant servers. They are often used in applications such as databases [Chang et al. 2006] or file systems [Ghemawat et al. 2003] to replicate important information and reduce the risk of data loss or corruption from a failed hard drive. The physical separation between processes necessitates a communication channel of some kind, such as the Internet, or a datacenter's local intranet. However, these networks can be unreliable as they typically use TCP/IP protocols, which are inherently asynchronous and prone to errors such as dropped or delayed packets [Cachin et al. 2011].

To maintain data consistency under these conditions, distributed systems often rely on a class of algorithms called consensus protocols [Burrows 2006; etcd Developers 2013–2022], which are able to reach agreement among some sufficiently large subset of participants despite some amount of failures. Unfortunately, these protocols are notoriously difficult to understand and easy to implement incorrectly [Gill et al. 2011; Gunawi et al. 2014; Meza et al. 2018]. Even when written by experts, bugs are sometimes introduced that can cause

disruptions to major online services [AWS Team 2011; Treynor 2014].

Testing and model checking can reduce, but never fully eliminate, the risk of these bugs because the search space is too large to cover exhaustively. The only way to completely guarantee correctness is formal verification, which produces a mathematical proof that all allowed behaviors of a program satisfy some property. However, manual pen-and-paper proofs are susceptible to invalid logical steps or overlooked cases [Berger et al. 2021; Cachin and Vukolic 2017; Momose and Cruz 2020; Ongaro 2015; Whittaker 2020]. A more reliable option is to use a mechanized proof assistant, such as Coq [Coq Development Team 1999–2022]. This provides an expressive dependently typed language for writing specifications and proofs along with a type checker to validate them (an extreme version of "well-typed programs cannot go wrong" [Milner 1978]). Recent years have seen an abundance of research on this topic and the development of many useful verification frameworks and techniques [Hawblitzel et al. 2015a; Krogh-Jespersen et al. 2020; Ma et al. 2019; Padon et al. 2016; Sergey et al. 2017; Shin et al. 2019; Wilcox et al. 2015]; however, these often struggle to scale to larger or more complex systems as the number of cases to consider becomes overwhelming.

Generally in computer science, when a problem is too complex to reason about, the solution is to find the correct abstraction that distills it down to only its essential elements. The research question addressed by this dissertation is *what does this abstraction look like for distributed systems*, with a particular focus on consensus protocols. The proposed solution is the novel atomic distributed object (ADO) model, whose main features are a tree-based data structure to represent temporary data inconsistencies, and a simple, but expressive atomic interface. The remainder of this section discusses shortcomings of

existing abstractions, the challenges in designing a new one, and a summary of the ADO model's contributions.

1.1 Distributed System Abstractions

Abstraction Layers When reasoning about distributed systems there are three main layers of abstraction to consider. On top is an application, which provides some client-facing interface for a replicated object (e.g., Chubby [Burrows 2006] or ZooKeeper [Hunt et al. 2010]). Under that is a distributed protocol like Paxos [Lamport 1998; van Renesse and Altinbuken 2015] or Raft [Ongaro and Ousterhout 2014], which manages replication and consistency. At the bottom is the network level, which implements the communication primitives used by the protocol and handles issues such as server crashes and network asynchrony. Each layer has different goals and challenges, so it is important to model the state of a distributed system in a way that suits the properties being proved.

For example, the purpose of an application like a distributed key-value store is to give the illusion of being a standard object and satisfy properties, such as that reading a key immediately after setting it returns the new value. Ideally, the proof of such properties should not need to consider network failures or even the existence of separate servers. Instead, one should be able to reason purely about the object's state and methods just as in a non-distributed setting.

An example of a protocol-level correctness property is that there always exists a latest committed snapshot of the replicated state from which all later versions of the state evolve. This is completely orthogonal to whatever application is built on top of the protocol, so

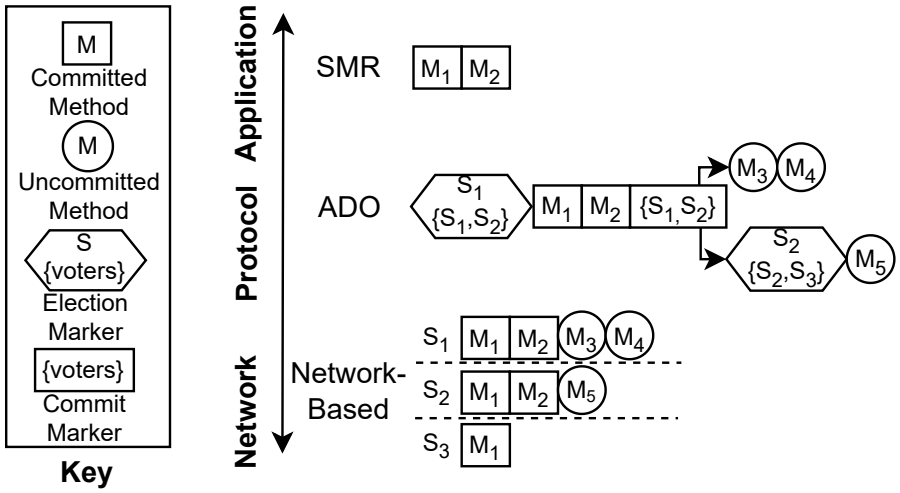


Figure 1.1: The spectrum of distributed system models. Each shows a snapshot of the same distributed state at a different level of abstraction. Network-based represents each server (S_1, S_2, S_3) as a separate object. SMR merges them into a single log and exposes only committed methods. The ADO model also merges the logs, but preserves uncommitted methods as branches of a tree. It also contain additional logical metadata in the form of election and commit markers that serve as evidence of the system's safety.

the details of the replicated state do not matter. Likewise, the same protocol might have many possible low-level implementations with different optimizations and communication patterns, but these details are also irrelevant as long as they do not change the high-level invariants that the protocol satisfies.

Finally, a network-level proof might show that a reliable broadcast function guarantees all participants will eventually receive a message. This layer is about providing efficient and correct communication primitives that the protocol layer can use as building blocks without worrying about the internal details.

There is no clear divide between different abstraction layers, but rather a continuous spectrum. Figure 1.1 shows some of the most commonly used models and how the ADO model fits into their hierarchy.

Network-Based At one end is the class of network-based models, which represent a protocol's behavior in terms of a set of servers passing messages over an abstract network. This is very flexible and can express any protocol along with all of its optimizations and extensions. It also closely mirrors the actual implementation, which means that properties proved about the abstraction are likely to hold for the implementation as well. Unfortunately, it fails to isolate network and protocol-level logic, which means one must consider both at the same time. This makes reasoning more challenging, and it also tightly couples the protocol and implementation, making proofs less reusable.

State Machine Replication On the opposite end of the spectrum is state machine replication (SMR) [Schneider 1990], which gives the illusion of a single, atomically-accessible object (represented by a log of committed commands) rather than a collection of individual servers. It hides internal communication details and the existence of intermediate inconsistent states behind a remote procedure call (RPC) interface. This makes it ideal for application-level reasoning, but much too abstract to prove anything at the protocol level. It is also unable to model applications that take advantage of, instead of hiding, their distributed nature by relying on certain features of their underlying protocols [Gray and Lamport 2006; Zhang et al. 2015].

Atomic Distributed Objects The goal of the ADO model is to fill the gap between these abstractions and target the protocol layer as well as systems that straddle the protocol-application border. Like SMR it models the distributed state as a single object with an atomic object-oriented interface. However, it unfolds RPC calls into three steps that correspond

to the general structure of most consensus protocols. Each step can fail, which exposes intermediate states where servers may temporarily disagree on the replicated state.

These partial failures are modeled by representing the state as a tree comprised of a sequence of committed methods with uncommitted branches. To enable protocol-level reasoning, the tree also contains logical markers that indicate how in-agreement the participating servers are at certain times. This provides vital information that allows one to reason about not just the current state of the system, but the history of how it arrived there, while still maintaining a clean separation from network-level details.

1.2 Challenges

As a protocol-level abstraction, the ADO model is able to reduce unnecessary complexity and more effectively support reasoning about optimizations and properties that are impractical in network-based or SMR models. The following are examples of important aspects of distributed systems that the ADO model can cleanly express, but that prior work has struggled to address.

Composition Modern distributed applications are typically not a single, monolithic system, but rather a collection of interacting components [Dean 2009]. When combined with the already intricate behaviors of individual systems, it is clear that attempting to reason about the entire system at once would quickly become hopelessly complex. Instead, this problem requires a compositional abstraction that allows one to reason about an individual component's internal behaviors in isolation as well prove properties about the

interactions between components. This also provides an additional level of modularity as a component can be verified once and reused by multiple applications.

Many verification frameworks simply do not support reasoning about multiple components. This is insufficient for practical applications, because even if two components are verified separately, an incorrect interface between them can introduce serious bugs that threaten the whole system [Fonseca et al. 2017]. The majority of the few frameworks that do handle composition describe the interactions at the network level [Krogh-Jespersen et al. 2020; Sergey et al. 2017]. This makes it difficult to scale to larger systems, but it also ties the proofs to a particular implementation. This can be a problem because different components may have varying performance and reliability requirements, so the application could be implemented by a heterogeneous collection of protocols. If, for example, a developer decided to swap one protocol for another for performance reasons, the proof would break even if they are both consensus protocols that satisfy the same high-level properties.

Reconfiguration In practical systems, the set of participating replicas may not be constant as old servers are taken down for maintenance and new ones are added to cope with increased load. This process is called reconfiguration, and it is necessary for realistic distributed systems, but it deeply interacts with a protocol's core invariants in a way that makes it difficult to verify. Adding or removing a server at the wrong time can easily compromise the consistency of the replicated data by allowing committed data to be overwritten, or even rendering the entire system inoperable [Gunawi et al. 2014]. The fundamental challenge is that reconfiguration changes the metadata that protocols rely on to achieve consensus (e.g., membership, quorum sizes), but it itself also depends on

consensus to ensure the changes are applied consistently.

This circularity creates subtle dependencies among different aspects of the protocol, such that it can be difficult even for experts to fully anticipate how reconfiguration influences the correctness properties. For example, Ongaro [2014] proposed a reconfiguration scheme for the Raft protocol that, despite extensive peer review and several implementations, was found to have a critical bug nearly a year later [Ongaro 2015]. A fix was proposed along with a loose sketch of its correctness, but no formal proof was given. In fact, there is extremely little prior work on formal verification of reconfiguration, which we expect is largely due to the lack of an appropriate abstraction to manage the complexity.

An additional challenge that a good protocol-level abstraction should solve is supporting general classes of reconfiguration schemes. As with consensus protocols, there are many implementations, but most follow common patterns that, if abstracted properly, could allow proofs to be easily reused.

Time Time is very important in distributed systems. Communication over a network is unreliable, so servers cannot simply wait forever for a response or the system might deadlock. Instead, there is typically some timeout after which they abandon the attempt and try something else. Servers are generally not assumed to have exactly synchronized local clocks so timeouts may occur at different times, which, if not handled carefully, can make it very difficult for servers to coordinate.

Temporal properties (e.g., a server eventually acknowledges a message) are often more challenging to prove than consistency because they require considering everything that could happen rather than just what has already happened in a given case. Many prior veri-

fication efforts simply ignore these properties, and those that do handle them often require a special temporal logic separate from what is used for consistency proofs [Hawblitzel et al. 2015a; Losa and Dodds 2020].

Generalized Failure Models Failures are inevitable in a distributed setting, so protocols must be prepared to handle them. How they do this depends on the failure model; i.e., the assumptions about what effects failures can have. For example, under the benign, or fail-stop, model [Cachin et al. 2011] servers may become unresponsive, but the byzantine model [Lamport et al. 1982] allows certain servers to actively work against the others. The former case only needs to guard against waiting indefinitely for a response from a crashed server, but in the latter servers cannot necessarily even trust the messages they receive.

It is not immediately clear that the gap between byzantine protocols and their benign counterparts can be bridged. The lack of trust between servers seems to require fundamental changes to the communication patterns, and indeed, early byzantine protocols such as PBFT [Castro and Liskov 1999] differ in many ways from their benign predecessors. Surprisingly, it turns out that these differences are not insurmountable and that, at its core, consensus works on the same basic principles regardless of failure model. Therefore, with some clever parameterizations, it is possible to unify the byzantine and benign cases into a single model [Lamport 2011; Rütti et al. 2010].

This generalized approach has not been applied in any mechanized verification work, likely because it is difficult to express in the standard network-based models where the implementation-level differences are more pronounced. This is an ideal problem for a protocol-level abstraction like the ADO model since it strips away these details, and instead

highlights the common features. The advantage of doing so is it allows one to prove a property once and for all and reuse it for a large variety of protocols.

1.3 Contributions

The primary contribution of this dissertation is a formal definition of the novel, protocol-level abstraction called the atomic distributed object (ADO) model, along with an empirical evaluation of the practical effectiveness of the model through case studies. Each study presents a version of the ADO model, implemented in Coq, as a novel solution to a previously unsolved challenge in distributed system verification.

- ADVERT (atomic distributed object verification toolchain) is an abstraction designed for reasoning about distributed applications that take advantage of protocol-level features such as partial failures, as well as the composition of such systems. This enables the verification of applications that prior work was either too restrictive to express, or insufficiently abstract to manage the complexity.

- ADORE (atomic distributed objects with reconfiguration) enables reasoning about dynamically reconfigurable consensus protocols in terms of abstract ADO state and atomic interface instead of asynchronous network events. This greatly simplifies proving correctness properties, which is essential because reconfiguration introduces many subtle dependencies that are only made clear when irrelevant details are stripped away. Prior to ADORE, no mechanized proofs of the safety of a consensus protocol handled reconfiguration.

- AdoB (atomic distributed objects for benign/byzantine consensus) extends the ADO model's capabilities with support for temporal reasoning and a common specification for both benign and byzantine consensus. The former makes it possible to prove that not only does a protocol never cause the replicated state to appear inconsistent, but it will, in fact, eventually converge on a consistent state. The latter allows one such proof to hold for an extremely broad class of protocols.

1.4 Organization

The remainder of the dissertation is organized as follows. Chapter 2 gives important background on consensus and existing distributed system abstractions. Chapter 3 presents an informal overview of the key features of the ADO model. Chapters 4 to 6 cover the Advert, Adore, and AdoB case studies, including formal definitions and evaluations. Chapter 7 provides a comparison with related work. Chapter 8 discusses future work and concludes. The following is a brief summary of each of the ADO variants.

Advert Advert defines an atomic semantics that faithfully captures the common high-level behaviors of consensus protocols, while abstracting away unnecessary protocol-specific details such as packet interleaving and quorum sizes. This allows one to write ADO-level specifications and proofs for an application that are completely independent from the specifics of its implementation.

In particular, we prove correctness properties about several distributed key-value store designs that support partitioning and replication via ADO composition. This includes a

"lock-free" design that composes two objects without any centralized coordinator. We also show a version of Two-Phase Commit [Cachin et al. 2011] with replicated resource managers that demonstrates how the additional details exposed by Advert enable reasoning about optimizations that cannot be expressed in an SMR-like model.

Finally, we prove a refinement between Advert and network-based specifications of several protocols including Single Paxos [Lamport 2001], Multi Paxos [van Renesse and Altinbuken 2015], Vertical Paxos [Lamport et al. 2009], CASPaxos [Rystsov 2018], and Chain Replication [van Renesse and Schneider 2004]. The Multi Paxos specification is formally linked to a C implementation using certified concurrent abstraction layers [Gu et al. 2018] and an executable binary is generated by a verified compiler [Leroy 2009].

Advert was originally published in Honoré et al. [2021a]. A Coq artifact is available on Zenodo [Honoré et al. 2021b].

Adore Adore supports proving correctness guarantees of consensus protocols with reconfiguration. In particular, it targets the safety of the challenging class of "hot" algorithms, which dynamically adjust the membership while also processing client requests. The reconfiguration scheme is quite general and admits a variety of implementations as long as they satisfy a few basic invariants.

We prove that Adore satisfies the key safety property that committed states are never lost or overwritten. This guarantee applies to any benign fault tolerant consensus protocol with a compatible hot reconfiguration scheme that refines Adore. The proof also brings to light subtle circularity issues that were not apparent in previous informal proof sketches, but we demonstrate that an elegant solution naturally arises from the ADO model's tree-

based state representation.

We also show the generality of the model by instantiating the generic reconfiguration scheme with several practical algorithms including Raft single-server [Ongaro 2014], Raft joint consensus [Ongaro and Ousterhout 2014], primary backup [van Renesse and Schneider 2004], and dynamic quorum sizes [Lamport et al. 2009]. Finally, we demonstrate the validity of the model by proving it is implemented by a network-based specification of a version of Raft, which can automatically extracted to executable OCaml code.

Adore was originally published in Honoré et al. [2022a]. A Coq artifact is available on Zenodo [Honoré et al. 2022b].

AdoB AdoB explores ADO-based temporal reasoning and bridging benign and byzantine models. It 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.

We prove that this generic model guarantees that the state is replicated consistently among the replicas and that, under partial synchrony assumptions [Dwork et al. 1988], progress is eventually made towards committing new commands. These proofs hold under both the benign and byzantine failure models and highlight the essential similarities and differences between them.

To demonstrate AdoB's relevance to actual protocols, we proved that a network-based specification of a variant of the Jolteon byzantine consensus protocol [Gelashvili et al. 2022] refines AdoB. We call this variant GenJolteon because it can be configured to handle

either a benign or byzantine setting by instantiating a few parameters. As with Adore, we also leveraged Coq's support for extraction to produce a verified, executable OCaml implementation of this protocol.

Funding This work was supported in part by NSF grants 1521523, 1763399, 1945541, 2019285, 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 NSF, DARPA, and NIWC Pacific.

Chapter 2

Background and Motivation

2.1 Consensus

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 in which each replica maintains a log of commands. Once a command is determined to be *committed*, replicas execute it locally. Therefore, to ensure consistency between the replicas, a consensus protocol must guarantee that they agree on the order of the committed commands.

2.1.1 Benign Consensus

There are many consensus protocols, but they broadly fall into categories depending on their failure models; i.e., what kind of errors the protocol is capable of handling while still functioning correctly. One of the simpler, but more common, failure models is the benign, or fail-stop, setting. This allows replicas to become unresponsive, but not to send invalid messages or otherwise behave incorrectly [Cachin et al. 2011]. The network is also

```
// Leader
elect() {
  self.time += 1;
  broadcast(Elect, self.time, self.log);
  self.votes := wait_for_votes();
  return isQuorum(self.votes);
}
local_update() {
  self.log.append(new_command(self.time));
  return true;
}
commit() {
  broadcast(Commit, self.time, self.log);
  self.votes := wait_for_votes();
  return isQuorum(self.votes);
}
// Replicas
handle_elect(ldr, time, log) {
  if (self.time < time) && (self.log.last.time <= log.last.time) {
    self.time := time;
    send(ldr, ElectAck);
  }
}
handle_commit(ldr, time, log) {
  if (self.time <= time) && (self.log.last.time <= log.last.time) {
    self.time := time;
    self.log := log;
    send(ldr, CommitAck);
  }
}
```

Figure 2.1: Benign consensus pseudocode.

typically assumed to be unreliable in the sense that it may delay, duplicate, reorder, or drop messages, but not corrupt their contents. The challenge of benign consensus is to reach agreement with the participation of only some subset of the replicas (called a *quorum*).

Different protocols have different methods of accomplishing this task, but they generally work by repeating three phases: election, local update, and commit (represented in pseudocode in Figure 2.1 as `elect`, `local_update`, and `commit`, respectively). The goal of the election phase is for a replica, known as a *candidate*, to get permission to add new log entries by confirming that its has all of the existing committed entries. This requires a

way to determine which of two logs is more "up to date". Replicas do not have access to a synchronized global clock, so instead, every command is assigned a logical timestamp and logs are compared by the timestamps of their last entries (e.g., Line 19 in `handle_elect`).[1]

A candidate solicits votes from the other replicas (in some protocols the candidate may also vote for itself), who decide whether to vote by comparing the candidate's timestamp and log to their own. If the candidate collects a quorum of votes then it becomes the leader and can be confident that, at the moment, it has the most up-to-date log among its voters. The size of a quorum varies by protocol, but it must be at least large enough to ensure that any two quorums must share at least one replica, so a simple majority is common. The importance of quorum overlap is discussed further in Section 2.1.3.

In the local update phase, the leader simply appends a new command to its log, typically at the request of an external client. Some protocols allow multiple consecutive local updates, but eventually the leader attempts to replicate the new entries in the commit phase.

Here again, the leader's goal is to share its log and convince at least a quorum of replicas that it is sufficiently up-to-date. This second check is necessary because in the time since its election, the leader may have been preempted by another, more recent leader, in which case the old leader's log cannot be trusted to contain all committed entries. If a replica determines that the leader has not been preempted, then it updates its own log to match the leader's. If a quorum do so then the new commands are now committed. Now the cycle repeats with a new election; however, some protocols simply continue with the same leader and use the previous successful commit as an implicit quorum of votes.

[1]For simplicity, the pseudocode assumes there is at most one entry per timestamp. If this is not the case, ties may be broken using the log length.

2.1.2 Byzantine Consensus

Benign failure assumptions are appropriate for controlled environments such as data centers, but in more open settings (e.g., blockchains) participants cannot always be trusted. Protocols that operate under this failure model are called byzantine [Lamport et al. 1982]. 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 lying about local state.

As with benign consensus, the protocol breaks down into three phases (Figure 2.2). The election and commit phases serve the same purpose as before, but an important difference is that replicas can no longer trust the leader or each other. For instance, they cannot be sure during the commit phase that the log that the leader proposes is the same log being proposed to everyone else. If it were not and the replicas simply accepted the different logs, a byzantine leader could cause the committed states to diverge.

Since no individual can be believed, the only way to gain trust is through a quorum. However, it is no longer sufficient to simply require that quorums overlap, as there is no guarantee the common replica is honest. Instead, operations now require a *super quorum* of voters, which must have at least one honest replica in common with every other super quorum. For example, if at most 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.

Aside from the larger quorums, `elect` and `commit` work mostly as before, broadcasting requests containing the leader's current log, and waiting to collect votes. However, as another consequence of the reduced trust between replicas, the second phase, formerly a

```
// Leader
elect() {
  self.time += 1;
  broadcast(Elect, self.time, self.log);
  self.votes := wait_for_votes();
  // NEW: isSQuorum = "super quorum"
  return isSQuorum(self.votes);
}
precommit() {
  self.log.append(new_command(self.time));
  // NEW: Include votes as evidence of successful election
  broadcast(PreCommit, self.time, self.log, self.votes);
  self.votes := wait_for_votes();
  return isSQuorum(self.votes);
}
commit() {
  // NEW: Include votes as evidence of successful pre-commit
  broadcast(Commit, self.time, self.log, self.votes);
  self.votes := wait_for_votes();
  return isSQuorum(self.votes);
}
// Replicas
handle_elect(ldr, time, log) {
  if (self.time < time) && (self.log.last.time <= log.last.time) {
    self.time := time;
    send(ldr, ElectAck);
  }
}
// NEW: Confirm that ldr has enough votes, and that log is safe to commit
handle_precommit(ldr, time, log, elect_votes) {
  if (self.time <= time) && (self.log.last.time <= log.last.time)
     && validate(elect_votes) {
    self.time := time;
    send(ldr, PreCommitAck);
  }
}
handle_commit(ldr, time, log, precommit_votes) {
  if (self.time <= time) && (self.log.last.time <= log.last.time)
     // NEW: Confirm that ldr did precommit
     && validate(precommit_votes) {
    self.time := time;
    self.log := log;
    send(ldr, CommitAck);
  }
}
```

Figure 2.2: Byzantine consensus pseudocode. Comments beginning with NEW mark changes from the benign case.

local update in the benign case, now also requires a super quorum of votes and is called the pre-commit phase. The purpose of this step is for a super quorum of replicas to attest that they would be willing to commit a particular command if the leader can prove that it has sufficient support. This rules out the scenario in which a byzantine leader proposes different logs during the commit phase.

In both the pre-commit and commit phases, the leader must demonstrate that it completed the previous round with the requisite number of votes by including them collected votes in their requests (hence the `self.votes` argument to `broadcast` on Lines 12 and 18). The recipients then independently validate the votes (Lines 32 and 40 in `handle_precommit` and `handle_commit`). To ensure that the votes are genuine and cannot be forged by malicious leaders, they are typically cryptographically signed. Section 2.1.3 discusses this and other necessary assumptions further.

Note that although the byzantine failure model subsumes the benign case, the larger quorum sizes and additional rounds of communication come with an increased performance cost. Therefore, benign consensus protocols are still preferable when appropriate.

2.1.3 Safety and Liveness

Safety There are two primary correctness properties that a consensus protocol should satisfy. The first is *replicated state safety*, or simply safety, which states that clients observe the committed commands in the same order regardless of which replica they contact. This implies that if two replicas commit a command in a certain slot, the prefixes of their logs

up to that slot are equal.[2]

The key to maintaining this property for both benign and byzantine consensus is to ensure that concurrent commit requests can be linearized (i.e., their effect on the replicated state appears as if they executed in a sequential order). This is why elections and commits require a (super) quorum of voters, and why quorums are defined such that any two quorums have a common honest replica. The common voter must have received one of the commit requests before the other, and because replicas only vote for requests with monotonically increasing timestamps, the only way both requests could succeed is if the one with the smaller timestamp arrived first. This fixes the ordering for these two requests, and by continuing this reasoning, we can do the same for every commit.

As previously mentioned, replicas must be able to trust the authenticity of the messages they receive. Therefore, byzantine replicas cannot be allowed to forge messages from other replicas or tamper with the content of a message. In practice, this is often enforced with cryptographic methods such as threshold signatures [Shoup 2000].

Liveness Safety, while necessary, is not sufficient to guarantee that a system is useful. For example, a trivial protocol that ignores all messages is vacuously safe, but will never commit any commands. Therefore, the second essential correctness property is *liveness*, which guarantees that meaningful progress (i.e., committing a new command) is eventually made. This is complicated by the fact that 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].

[2]Certain protocols allow a slot to be committed while earlier entries are still undecided, but this property still holds once the gaps are filled in.

Despite this impossibility result, all is not lost if we simply introduce a few assumptions that can reasonably be expected to hold in practice. Unless otherwise specified, the following are required for both benign and byzantine protocols; however, note that none of the following are necessary for safety.

- There exists at least a *quorum of non-faulty replicas* that never crash for the benign case and a *super quorum of honest replicas* for the byzantine. For a typical benign majority quorum, this means at most f out of $2f + 1$ replicas may crash. Likewise, for the standard definition of a super quorum this means less than $1/3$ of replicas can act maliciously. We assume faulty, non-faulty, honest, and byzantine replicas are arbitrarily fixed in advance, but replicas are not aware of these assignments.

- 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 honest, 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 honest, non-faulty replica has a turn.

- Honest, non-faulty replicas follow a *productive strategy*; i.e., they perform operations in a timely manner whenever they are able. For example, an honest leader will attempt to commit new log entries after creating them within some finite time.

It is not difficult to guarantee liveness after GST as we can assume a (super) quorum of honest, non-faulty replicas will be active. Combined with the fair election rotation we

know that eventually an honest, non-faulty leader will be elected, at which point a new command can easily be committed. The challenge is to ensure that the system does not reach deadlock before this leader can be elected, for example by entering a state where no replica can be elected leader.

To avoid this, replicas maintain local timers that reset after every election. If they do not hear from the leader in that time they broadcast a timeout message. 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 advances its logical timestamp and prompts the next leader to begin an election. This ensures that even if no progress is made on committing new commands, replicas are constantly advancing to new rounds and will not become stuck before GST.

2.1.4 Protocol Examples

Figures 2.1 and 2.2 give a high-level overview of the basic elements of consensus protocols, but they omit many implementation details. This section fills in some of these gaps by summarizing a few popular consensus protocols for both the benign and byzantine settings.

Paxos Paxos [Lamport 1998, 2001] is a classic benign consensus protocol that has inspired many variations [Gafni and Lamport 2003; Lamport 2006; Lamport et al. 2009; Malkhi et al. 2008; Rystsov 2018]. The original version is only capable of reaching consensus on a single value, but it is extended to a log of values by Multi Paxos [van Renesse and Altinbuken 2015]. Unless otherwise specified, we use Paxos to mean Multi Paxos and write Single Paxos for the original.

As in Figure 2.1, Paxos begins with a kind of election (called the *prepare* phase), in which a leader with a sufficiently up-to-date log is chosen. The candidate broadcasts its request with a new logical timestamp (called a *ballot number*) and potential voters compare it against the largest timestamp they have observed thus far; however, rather than having replicas compare their logs against the candidate's, the candidate collects its voters' logs and chooses the latest.

The latest log can either be decided entry-by-entry, where, for each slot, the candidate chooses the one with the largest timestamp, or more holistically by comparing the timestamp of just the last entry and the log length. The former makes sense when the log entries are independent values, but, in the case where they are commands for a state machine, there is a dependency between consecutive slots that should be preserved by using the latter option.

Upon receiving a quorum of votes, the candidate becomes the leader and adds new commands to its log. Note that the log may also include uncommitted commands from previous rounds. The leader then attempts to commit these as well as its own commands in the commit (or *accept*) phase. Logically, this is achieved by broadcasting the new log to the replicas, who update their own logs after confirming the leader's timestamp is still the most recent they have seen. In practice, sending the entire log over the network is inefficient, so various optimizations are employed to reduce the packet size.

One option is to send each uncommitted command individually along with its intended position in the log. If a replica find it is missing the previous entry, it requests it from the leader, thereby filling any gaps. In this approach, it is possible that only some of the new commands are committed if the leader crashes or is preempted before finishing. The risk

of this can be reduced by batching some or all of the uncommitted commands into a single message. The ideal choice is application-specific and depends on the size and frequency of the commands. Regardless, the end result is that some prefix of the new commands is committed by replicating them across at least a quorum of replicas.

Raft Raft [Ongaro 2014; Ongaro and Ousterhout 2014] was designed to be more easily understood than Paxos and to more precisely define certain practical aspects of the protocol that Paxos leaves unspecified. For example, Paxos describes how elections work, but does not dictate when they should begin. This is irrelevant for safety, but if every replica were constantly competing for leadership it could hinder liveness by preventing any leader from running long enough to commit new commands. Raft solves this by assigning each replica a randomized timeout, only after which does it begin an election. Active leaders also send periodic *heartbeats* that reset the election timers.

The primary difference between Paxos and Raft elections is that Raft candidates propose their own logs instead of collecting them from their voters. Raft also requires that logs are compared using the timestamp of the last entry, with the log length as a tie-breaker. Although this shifts the burden of the log comparison from the candidate to the voters, the end result is the same: the leader's log is guaranteed to be the most recent among a quorum of replicas.

Raft leaders replicate their new log entries by tracking what entries each replica is missing and sending only the necessary suffix of the log. If a leader successfully commits its commands and receives a quorum of acknowledgements, it continues accepting client requests for new commands without another election.

Another area where Raft is much more explicit than Paxos is *reconfiguration*, which is the process by which participating replicas can be added or removed. This is essential in practice as replicas may crash or be brought down for maintenance. Paxos treats this as an orthogonal extension, but Raft defines it as a first-class operation of the protocol. This turns out to be a surprisingly challenging feature to implement correctly with subtle implications for a protocol's safety. Reconfiguration is one of the primary topics of Chapter 5, so we defer an explanation of Raft's reconfiguration scheme to Section 5.1.

HotStuff and Jolteon One of the first byzantine consensus protocols was PBFT [Castro and Liskov 1999], but the more recent HotStuff [Yin et al. 2019] and Jolteon [Gelashvili et al. 2022] protocols make certain interesting implementation choices that influenced the ADO model's design (see Chapter 6).

HotStuff (in particular, a two-phase variant [Bravo et al. 2020]) and Jolteon follow the usual sequence of operations: election, pre-commit, commit. In order to overcome the lack of trust between replicas, leaders include *quorum certificates* (*QCs*) in their requests as evidence that the operation is approved (similar to `self.votes` in Figure 2.2). A *QC* is a collection of a super quorum of votes, containing the identity of the voter, their current timestamp, and the *QC* for its latest log entry. Votes are cryptographically signed and combined using threshold signatures, which ensure that *QCs* cannot be forged. By attaching a *QC* to every request the replicas build up a trusted chain of evidence that guarantees byzantine replicas cannot break the safety guarantees.

Once a *QC* is formed the round ends and the next leader begins (leaders are decided in advance using some deterministic scheme such as round-robin). However, rather than a

Paxos or Raft-style election where the leader solicits votes, in HotStuff and Jolteon any replica can send the previous round's *QC* to the next leader. The effect is the same, i.e., a successful leader is guaranteed to have the most recent state snapshot among at least a super quorum of replicas.

The leader may then attach a new entry to its log and attempt to pre-commit it by proposing it to the other replicas. As in a commit round, the replicas check that the log is sufficiently up-to-date and that the accompanying *QC* is valid. If it is satisfied, it sends back its vote, but *does not* yet update its own log because it cannot know if the leader is trustworthy. If the leader receives a super quorum of votes, it forms a pre-commit *QC*, which it then uses in the commit phase to convince the replicas that it is safe to update their logs. Note that, unlike Paxos and Raft, HotStuff and Jolteon only allow one new command per round.

Under good conditions, the chain of *QCs* continues to grow, but if a round ends in timeout either due to network failures or malicious replicas, there is a break in the evidence chain. The solution is to construct a *timeout certificate* (*TC*). This serves a similar role as a *QC*, but instead of containing a super quorum of votes, it contains a super quorum of timeout messages, each of which contains the timed-out replica's latest *QC*. If a *TC* can be formed it guarantees no *QC* can also be formed for the current round, which fills in the gap and assures the replicas it is safe to move to the next round.

Upon timing out, a replica cannot know if the leader is honest or responsive, so it broadcasts its timeout message to all other replicas, and any replica can construct a *TC* once it sees enough messages. Once constructed, the *TC* is then sent to the next leader, who then begins the next round by selecting the most recent *QC* among those in the *TC*.

To prove that it did indeed choose the most recent, it also includes the *TC* in its requests for other replicas to independently verify.

Some versions of HotStuff and Jolteon additionally implement an optimization called pipelining that takes advantage of the close correspondence between the pre-commit and commit phases to combine them. A round in a pipelined protocol has only two phases: an election, followed by a commit phase. However, the danger of a malicious leader still exists, so a command is not actually considered committed until there are two consecutive committed commands (a two-chain commit in blockchain terminology). Essentially, every commit phase is simultaneously a pre-commit for the current round and a commit for the previous round.

2.2 Distributed System Abstractions

In order to verify properties of a distributed system, one needs a formal model to represent the possible behaviors. These models can vary in their level of abstraction (Figure 1.1), which affects what properties are simple, difficult, or even impossible to prove. This section discusses two of the most common abstractions, their strengths and weaknesses, and where the ADO model is necessary to cover their shortcomings.

2.2.1 State Machine Replication

The purpose of consensus is to provide the illusion of single object that is, in reality, spread across multiple replicas. State machine replication (SMR) [Schneider 1990] commits to this goal by modeling a system as a single, atomically-updatable log of commands. Clients

```
// SMR
return rpc_call(["put", "a", 1]);
```

```
// Network
self.time += 1;
broadcast(Elect, self.time, self.log);
self.votes := wait_for_votes();
if !isQuorum(votes) { return FAIL; }
self.log.append((["put", "a", 1], self.time));
broadcast(COMMIT, self.time, self.log);
self.votes = wait_for_votes();
if isQuorum(votes) { return OK; } else { return FAIL; }
```

Figure 2.3: Pseudocode representing the client-facing interfaces for updating a distributed key-value store in an SMR and (benign) network-based model.

extend the log through an opaque remote procedure call (RPC) [Tanenbaum and van Steen 2006; Wollrath et al. 1996] interface, which internally relies on a consensus protocol.

This model hides the internal communication details and the existence of intermediate states with uncommitted commands, which makes it very convenient for applications that are not concerned with the inner workings of the distributed system. For example, consider a distributed key-value store with a `put` command to create a new key-value mapping. From a client's perspective, `put("a", 1)` is an atomic action that either immediately updates the state or times out and fails (`SMR` in Figure 2.3). Internally, however, this illusion is achieved by a sequence of steps, each of which may fail and restart several times before the final result is reached.

Hiding these intermediate states certainly simplifies the model, but it also makes it impossible to reason about anything that happens "under the hood". For example, one cannot use SMR to prove the safety of a consensus protocol because it does not have any concept of separate replicas with their own logs, so it is meaningless to talk about agreement among a quorum.

Even if one is interested only in the high-level behaviors of a distributed application rather than the protocol that implements it, there can be advantages to opening the black box and exposing certain intermediate steps. Temporary failures and inconsistencies are inevitable in distributed settings, and a model that hides their existence rules out certain applications and optimizations that take advantage of them. These are discussed further in Chapter 4.

2.2.2 Network-Based Models

At the other end of the abstraction spectrum are network-based models, which more closely follow the real implementations by treating the system as a set of replicas that maintain their own local states and communicate over some abstract network. The model may take a simplified view of the network, for example by assuming messages are never duplicated, but the distinguishing feature from SMR is that communication is asynchronous and non-atomic.

For example, consider `put("a", 1)`, which is no longer an atomic operation, but a long sequence of steps (`Network` in Figure 2.3). The `broadcast` operation marks the point where the request is sent, but it may not be received until potentially much later. Therefore, between sending the request and receiving the acknowledgements in `wait_for_votes`, another client might call `put("a", 2)`. This broadcasts a new request that can interleave with the previous one and may even be delivered before it.

This is a real possibility that consensus protocols must be prepared to handle, so it makes sense that a model for verification should also take it into account. However, doing

so naïvely can quickly create an intractable situation as various network failures multiply and interleave, creating an explosion in the number of cases to consider. The key to designing an abstraction that can scale effectively while still being faithful to the protocols it models is to draw a distinction between protocol and implementation-level details.

For instance, it is a basic requirement of a consensus protocol that it handle the case where a request is not accepted by all replicas. However, why this happens, whether because a message was delayed, or a replica crashed, is purely a technicality of the network and is irrelevant to the protocol. Network-level models are ideal for checking a high-level model against a specific implementation, but they unnecessarily complicate proofs of high-level properties like safety by mixing protocol and implementation-level logic.

Chapter 3

Atomic Distributed Object Overview

3.1 Inspiration

The distributed setting has much in common with the shared-memory concurrent world, in which multiple threads run simultaneously on the same machine. Therefore, when designing the ADO model we drew inspiration from concurrent object models, and the push/pull model [Gu et al. 2016, 2018] in particular, because it is compositional and has an interface that maps nicely onto the three-phase design of many consensus protocols. This section summarizes how we transformed ideas from the push/pull model into the ADO model by identifying the key differences between concurrent and distributed objects. We use the pseudocode of a simple FIFO `Queue` in Figure 3.1 as a running example of an ADO. Section 4.3 explains the notation further, but for now it is sufficient to understand that this represents a queue with atomic methods that is implemented by a replicated `Vector` (resizable array).

```
ADO Queue {
  shared data : Vector[ℤ] := [];
  method enqueue(val) { this.data.append(val); }
  method dequeue() {
    if (this.data.length > 0) {
      val := this.data.pop(0);
      return Some(val);
    } else { return None; }
  }
}
```

Figure 3.1: Pseudocode of a FIFO Queue implemented as an ADO.

Push/Pull Basics The push/pull model represents an object's state as a logical history of the methods called up to that point (e.g., $\mathsf{enqueue}(1) \bullet \mathsf{enqueue}(2)$ represents the queue $\{1, 2\}$). The concrete value can be recovered by replaying the methods in the history, but for reasoning purposes it is convenient to remember the steps that led to the current state. Note that this is related to, but distinct from, the log of commands stored by replicas in many consensus protocols. Those represent runtime states that are actually stored in memory or on disk, but the push/pull method history is a logical abstraction that may or may not have a corresponding physical log in the implementation. For example, instead of a log of commands, an object may simply maintain the latest state and apply updates in-place (e.g., CASPaxos [Rystsov 2018]).

In order to interact with an object, the push/pull model allows clients to apply one of the object's methods (e.g., enqueue) or use two special operations for managing concurrent access: `pull` and `push`. To avoid data races, a client first calls `pull`, which creates a local copy of the shared state and takes ownership of the object. This also locks the state so that, during this time, other clients cannot access the object. The client then applies the method locally with `invoke`, which appends a new method to the copy of the history. Finally, it

commits the change by calling `push` to copy back the updates and release ownership. Note that there is a distinction between a *method invocation*, which only locally adds a method to the history, and a *method call*, which makes the effect globally visible by performing the entire sequence of `pull`, `invoke`, `push`.

Distributed System Challenges The push/pull model provides a straightforward framework for designing concurrent objects and reasoning about their atomicity, but there are some aspects that are clearly inadequate for handling distributed objects. For one, without atomic shared memory primitives like fetch-and-increment or compare-and-swap, a distributed replica cannot atomically claim exclusive ownership of an object. Instead, the best that can be achieved is a preemptible ownership that must be reconfirmed before committing an update (similar to the load-linked/store-conditional instructions).

Second, the potential for network failures mean that updates are not guaranteed to succeed. In fact, because in consensus success is determined by reaching agreement among a quorum, it is possible for a method call to partially fail, updating the logs of some non-quorum subset of replicas. These methods are not committed, so they are not guaranteed to persist, but they may still influence later operations.

The ADO model addresses these problem by keeping the push/pull model's three-step structure, but adjusting the behavior of each step. First, to represent network errors, `pull` and `push` may fail at any time. The precise reason for the failure is not modeled because it is not important for protocol-level reasoning. Second, `pull` no longer grants a replica exclusive ownership of the object, but instead marks it as only a temporary leader. If, between calling `pull` and `push`, another replica calls `pull` and succeeds, the original

leader's push will fail because its permission to update the state has been revoked.

Finally, to support partial failures, the logical history is changed to a logical tree of methods. Each node of the tree represents a method from some replica's log, and a fork in the tree represents a point where a partial failure created a transient disagreement. One can imagine constructing this tree by overlaying every replica's log, aligning the elements when they agree, and branching out where they do not. By analogy with memory models we refer to nodes of the tree as *caches* because they may contain volatile state that is waiting to be flushed to persistent storage (i.e., committed by a quorum of replicas). The *cache tree* keeps track of committed caches and ensures that they form a linear path through the tree, which guarantees the consistency of the replicated data.

3.2 State and Operations

3.2.1 Cache Tree

The central data structure of the ADO model's state representation is the cache tree, which models the current and previous local states of every replica in the system. Caches are classified into different types to represent different events. An election cache (*ECache*) indicates that a successful election occurred, a method cache (*MCache*) records that a method was invoked, and a commit cache (*CCache*) is created when a method is committed.

Every cache stores the identity of the replica that created it along with a logical timestamp. Caches may also store additional metadata depending on what properties one wishes to prove. For example, Adore (Chapter 5) and AdoB (Chapter 6) are designed for

proving the safety of consensus, so the caches remember the quorum of replicas that voted for them. Advert (Chapter 4) is targeted at slightly higher-level properties where this information is unnecessary, so it is omitted.

Dependencies between caches are represented by their parent-child relation. For example, a *CCache* marks all of its *MCache* ancestors as committed. One can reconstruct a replica's local log by following a particular branch of the tree from the root to a leaf, collecting all of the *MCaches* and ignoring *ECaches* and *CCaches*, which are purely logical markers. This transformation from cache tree to local logs is the basis of the refinement proofs between low-level network-based protocol specifications and the ADO model (see Sections 4.5.2, 5.5, and 6.5).

A fork in the tree represents a point where a network failure caused some replicas' local logs to temporarily diverge. However, recall that replicated state safety ensures that there exists a common prefix of committed commands shared by at least a quorum of the replicas' logs. The corresponding property for cache trees guarantees that, given any two *CCaches*, one is an ancestor of the other. In other words, every *CCache*, and therefore every committed method, lies on the same linear path through the tree, which implies the existence of a common prefix of committed methods.

In addition to the cache tree, the ADO state tracks the largest logical timestamp observed by each replica. As in the consensus pseudocode (Figures 2.1 and 2.2), this is used to filter out stale operations and ensure that log entries follow a strictly increasing total order.

3.2.2 Atomic Interface

Like the push/pull model, the ADO model provides three atomic operations for updating the cache tree. Each operation corresponds to one of the three consensus phases and is associated with a type of cache.

Pull Like the election phase, the purpose of `pull` is to elect a leader with a unique timestamp and an up-to-date state snapshot. In a consensus protocol, this involves broadcasting a request and waiting for a quorum of responses, but this view is both non-atomic (other events may occur between the broadcast and the responses) and exposes too many low-level details. Instead, the ADO model abstracts over all of the network communication details by assuming the existence of a logical *oracle* that nondeterministically returns either a successful or failed outcome.

One can think of this oracle as an omniscient observer that watches all network traffic until it knows that enough replicas have accepted or rejected a request. On success, the oracle chooses a unique timestamp for the leader and selects a cache to serve as its *active cache*. The active cache represents the up-to-date snapshot that the leader will build on during its term. If there are multiple equally up-to-date options, the oracle may choose one arbitrarily. This models the fact that a leader's choice of log may depend on which replicas reply to its request.

A failed `pull` represents a candidate that did not receive a quorum of votes, so no cache is added to the tree; however, it may still update the timestamps of some replicas. This could have a significant effect on the system because if enough replicas increase their timestamps, the previous leader may no longer have enough supporters to successfully

commit its methods.

Invoke Method invocation models the local update of benign consensus, or the pre-commit phase for byzantine. Depending on the failure model, the behavior is slightly different, but we will show in Chapter 6 how it can be generalized to work for both. For the moment, we consider only the benign case for simplicity.

As `invoke` is a local operation, there is no need for an oracle to decide the result. It simply creates an *MCache* and adds it to the leader's active cache, which sets the new cache as active. A leader may `invoke` several methods, each of which continues to append *MCaches* to the same branch. These methods are currently only in the leader's local log and have not yet been replicated.

Push The `push` operation attempts to commit all of the methods on the leader's active branch (the ancestors of the active cache). Like `pull`, an oracle hides the communication and possible network failures by nondeterministically deciding the outcome. On success, the oracle selects one of the newly created *MCaches* and adds a *CCache* after it, which commits all of the *MCache* ancestors. The reason the oracle is allowed to choose an *MCache* other than the last one is to cover certain protocols that also allow this behavior.

Suppose a leader has appended three new log entries and is preparing to commit them. In order to do so, it must decide how to share the entries with the other replicas. One option is to send all three entries in a single message, in which case either all or none of the entries will be received. However, it may instead choose to send each entry individually, in which case network failures may cause only some to be delivered. As we only consider

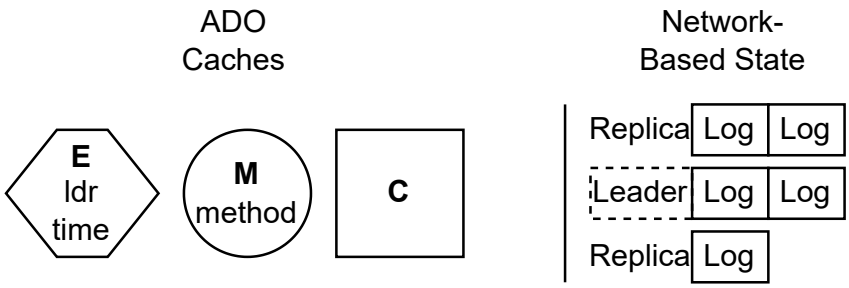
protocols that do not allow gaps in their logs, a replica that misses one entry will reject all of the following entries as well (e.g., if a replica receives the first and third entries, but not the second, it will only accept the first). Therefore, to model this case, the oracle chooses an arbitrary prefix of the *MCaches* to commit.

Note that, while `push` requires at least one *MCache* from the current leader in order to succeed, it may simultaneously commit *MCaches* from previous rounds as well. For example, suppose a leader has invoked a method, but before it manages to commit it, it is preempted by another replica's `pull`. The new leader's active cache will be set to the most up-to-date cache, which could be the previous leader's uncommitted *MCache*. The new leader then invokes its own methods, and because they are all on the same branch, a *CCache* will commit both the new and old leaders' *MCaches*.

Once a *CCache* is added to a branch, all sibling branches are effectively dead. The *CCache* is currently the most up-to-date cache in the tree, so a subsequent `pull` is forced to select it as the active cache, which ensures that all future *CCaches* descend from it.

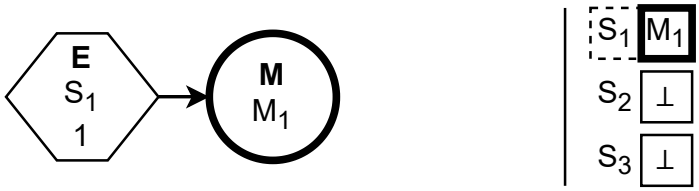
3.2.3 Examples

Figure 3.2 shows an example of a cache tree growing under good conditions with the corresponding network-based state of a Paxos or Raft-like protocol on the right. Each type of cache is represented by a different shape: hexagons for *ECaches*, circles for *MCaches*, and squares for *CCaches*. Some details are omitted for simplicity, such as the replicas' current timestamps and the *MCache* and *CCache* timestamps. The example considers a system with three replicas (S_1, S_2, and S_3) and each starts with an empty log.

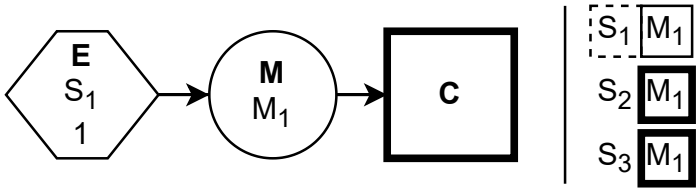


(a) The different cache types are represented on the left by different shapes. The network-based state on the right consists of each replica and its local log. The current leader is marked with a dotted border.

(b) `pull` elects S_1 and creates an *ECache*.

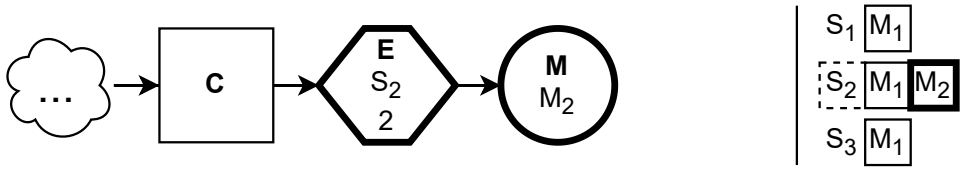


(c) `invoke` adds an uncommitted *MCache*.

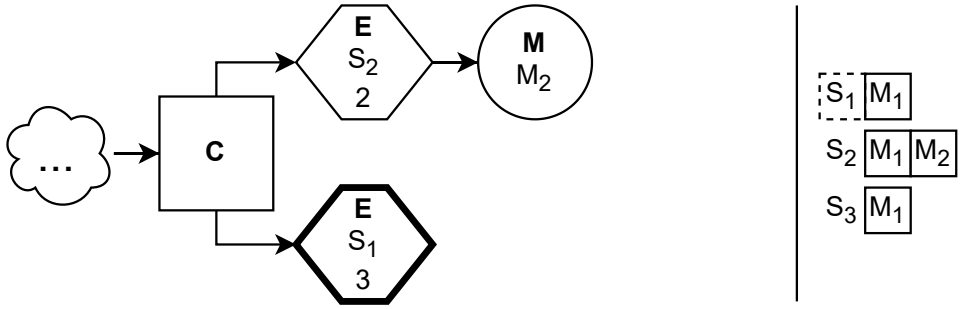


(d) `push` commits the method and creates a *CCache*.

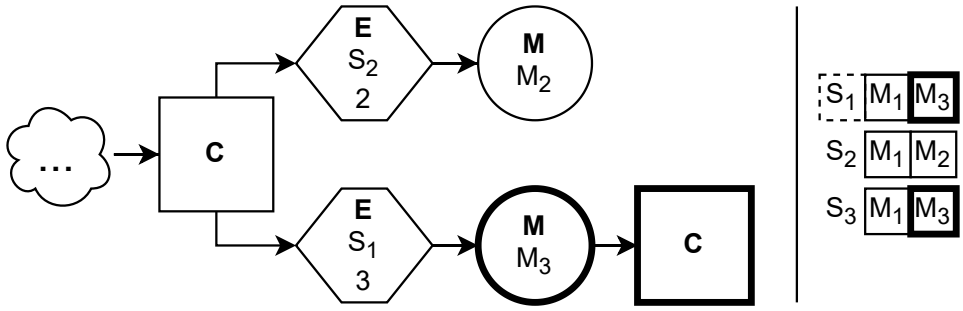
Figure 3.2: A cache tree's evolution in the ADO model without failures. Newly created caches are marked with a thick outline.



(a) S_2 is elected and invokes M_2. The cloud symbol abbreviates the *ECache*, *MCache* prefix from Figure 3.2.



(b) Before S_2 commits M_2, S_1 calls `pull` and succeeds, creating a fork.



(c) S_1 invokes and commits M_3, making S_3's branch unreachable.

Figure 3.3: A fork in the cache tree.

In the first step S_1 calls `pull` and succeeds, creating an *ECache* in the cache tree and marking itself as the leader on the network side (indicated by the dotted outline). It then invokes method M_1, which creates an *MCache* and adds an entry to its local log. Finally, S_1 commits M_1 with `push`, which replicates the entry to S_2 and S_3.

Under normal conditions, the tree may continue growing linearly in this way. For example, S_2 might call `pull` and then invoke M_2, which would add an *ECache* and an *MCache* directly after the *CCache*. Figure 3.3 demonstrates what happens if S_2 is preempted before it has a chance to commit M_2. Suppose that before S_2 calls `push`, it crashes. Eventually, S_1

times out and begins a new election and succeeds. The only replicas that can vote for S_1 are S_1 itself and S_3, neither of which has M_2 in its log yet. Therefore, the most up-to-date log contains just M_1, and the corresponding cache is the *CCache*, so the S_1's *ECache* creates a fork in the tree at this point.

From here, S_1 can proceed as normal, invoking and committing M_3 by replicating it among itself and S_3. Note that, although S_2's log still contains M_2 instead of M_3 in the second slot, because M_3 is in a quorum of logs (S_1 and S_3), any outside observer of the system is guaranteed to see M_3 instead of M_2. M_2 is stale state that will eventually be overwritten if S_2 recovers from its crash. In the cache tree, this means the branch with M_2 will never grow any further.

3.3 Advantages

Simplicity Compared to network-based models, one significant advantage of the ADO model is it abstracts away the details and complexities of network-based communication. Operations either succeed or fail immediately, significantly reducing the number of outcomes to consider. This allows verification efforts to scale to more complicated systems because one only has to handle complexity from one abstraction layer at a time.

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, the primary safety property can be expressed as an intuitive structural invariant about the tree, rather than an implicit relation between log prefixes. Likewise, temporary inconsistencies appear as forks in the tree, which makes it simple to find the point where the replicas diverged. Merging

the replicated state into a single data structure makes it much easier to visualize, which is useful for both formal and informal reasoning.

Expressiveness Despite hiding many network-level details, the ADO model is also much more expressive that SMR. By unfolding the RPC operation into separate `pull`, `invoke`, and `push` steps, it exposes partial failures, which are essential for proving protocol-level properties like safety and liveness. Without these intermediate steps, it is meaningless to even discuss safety because all methods are already assumed to be committed.

The ADO interface is also useful for application-level reasoning because it allows for more flexibility in defining method call patterns. Partial failures still exist in SMR, but they are hidden by automatically re-running an operation until it succeeds. This is also possible in the ADO model, but, for example, one could also define a method call that only tries once, or that performs some cleanup operation on failure (see Section 4.3.1 for more on these patterns). Having these additional options opens up the possibility for verifying a wider range of distributed applications.

Generality The ADO model provides a uniform, generic interface for consensus that can be implemented by many different protocols. For example, as far as it is concerned, there is no distinction between a Paxos or Raft election. Any differences are hidden by the cache tree and the oracle, which leaves only the core essence captured by `pull`. This means challenging proofs can be performed once at the ADO level and then reused for many protocols by proving a refinement relation.

The ADO model as presented thus far is actually a general framework for protocol-level

abstractions. The state is always represented by a cache tree and the `pull`, `invoke`, and `push` operations have the same basic effect, but the contents of the caches can be augmented depending on what information is required. This allows it to adapt to a variety of use cases, from compositional application-level reasoning (Chapter 4), to proving safety with reconfiguration (Chapter 5), to reasoning about liveness under a unified benign-byzantine failure model (Chapter 6).

Chapter 4

Advert: Atomic Distributed Objects for Composition and Partial Failures

This chapter describes the Advert abstraction and how it supports modular reasoning about distributed applications at different abstraction layers (Figure 4.1). It begins with a summary of the challenges of working with distributed objects (Section 4.1), followed by a formal presentation of the semantics of the Advert variant of the ADO model (Section 4.2). The following sections discuss different levels of end-to-end distributed system verification: constructing and reasoning about individual ADOs (Section 4.3), composing ADOs (Section 4.4), and verifying executable implementations against ADO specifications (Section 4.5). Finally, Section 4.6 evaluates the proof effort and performance of the verified code, and Section 4.7 summarizes the results.

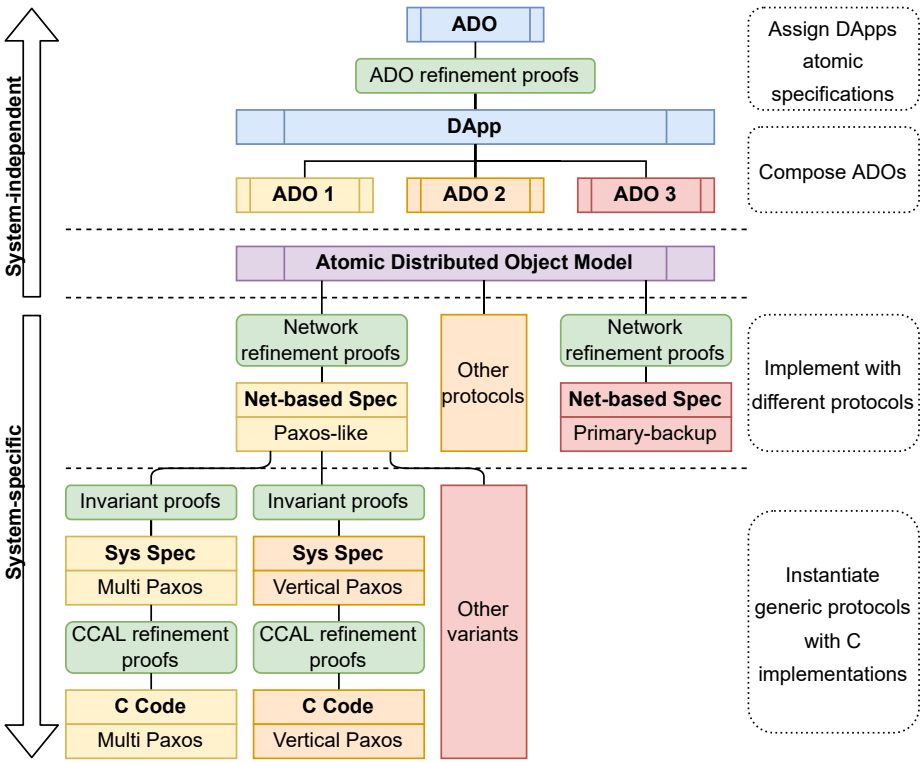


Figure 4.1: Different layers of reasoning with ADVERT, ranging from the composition of abstract objects to C code refinement proofs.

4.1 Motivation

Distributed applications are difficult to reason about because they are built from multiple distributed components, each with their own internal communication and failure handling methods. Modeling these components as self-contained objects with a well-defined boundary between their private state and public interface allows for much more modular reasoning. There are, however, fundamental differences between concurrent and distributed objects [Waldo et al. 1994] that must be taken into account.

The most significant of these differences is the possibility for methods to fail. Failures in distributed systems are much more common than in shared-memory settings [Gill et al. 2011; Gunawi et al. 2014; Meza et al. 2018]. Therefore, partially committed states are

inevitable, which, despite being transient, can influence later committed states. The SMR model treats these intermediate states as internal details and hides them from clients by waiting to reply until the system settles and consensus is reached.

This works well for the common case, but it can be overly restrictive. For example, if a call fails, rather than retry, an application might prefer to abort and execute a different operation. Exposing the individual steps of a method call along with the resulting intermediate states gives applications more freedom to choose how to handle failures. Section 4.3.1 describes some common method-calling patterns for various application requirements and shows how each is supported by Advert.

Certain systems even use partially committed states in order to optimize performance, but SMR is unable to accurately capture their behaviors. As a simple example, an application can execute a "fast read" by only running the election phase and skipping the commit step that guarantees the returned state is consistent. This is a kind of speculative execution so the application must implement some type of rollback mechanism, but, if the risk is low relative to the time saved, it could be a valuable optimization. Another interesting case is consensus combined with distributed transactions, where partially committed states can be used as hints to speed up transaction decisions (see Section 4.4.2).

4.2 Advert Formal Semantics

This section provides formal definitions of the types and operations of Advert, along with their semantics. For the most part these follow their intuitive descriptions from Section 3.2.

$$
\begin{aligned}
\mathit{Cache} &\triangleq \mathit{ECache}(\mathbb{N}_{nid} * \mathbb{N}_{time}) \\
&\mid \mathit{MCache}(\mathbb{N}_{nid} * \mathbb{N}_{time} * \mathit{Method}) \\
&\mid \mathit{CCache}(\mathbb{N}_{nid} * \mathbb{N}_{time}) \\
\mathit{CacheTree} &\triangleq \mathbb{N}_{cid} \rightharpoonup \mathbb{N}_{cid} * \mathit{Cache} \\
\mathit{TimeMap} &\triangleq \mathbb{N}_{nid} \rightharpoonup \mathbb{N}_{time} \\
\mathit{ActiveMap} &\triangleq \mathbb{N}_{nid} \rightharpoonup \mathbb{N}_{cid} \\
\Sigma &\triangleq \mathit{CacheTree} * \mathit{TimeMap} * \mathit{ActiveMap} \\
\mathit{ADO} &\triangleq \Sigma * (\lambda\rho\lambda\mathscr{R}.\, \mathit{Method} \rightharpoonup (\rho \to \rho * \mathscr{R}))
\end{aligned}
$$

Figure 4.2: ADVERT state definitions.

State Figure 5.3 defines the type of the global system state, Σ, to be a triple of a cache tree (*tree*), a partial map that stores the largest timestamp that each replica has observed (*times*), and a partial map that that marks each replica's *active* cache (i.e., the replica's view of the most up-to-date state). We use the notation $\mathit{name}(\mathit{st})$ to represent extracting one of these fields (e.g., $\mathit{tree}(\mathit{st})$ returns the first element). An ADO consists of this global state and a method interface. A method interface is a partial map from *Method* to method bodies, which are functions parameterized by the types ρ of the replicated state (e.g., `Vector[`$\mathbb{Z}$`]` in `Queue`) and $\mathscr{R}$ for the return values. *Methods* are simply identifiers that represent the name and parameters of the method to call. For example, `Queue`'s method interface includes the *Methods* `dequeue()`, `enqueue(1)`, `enqueue(2)`, and so on.

A *Cache* is either an *ECache*, *MCache*, or *CCache*. Every cache carries the node ID (*nid*) of the replica that created it as well as a logical timestamp (*time*). *MCaches* additionally contain a *Method*. The *CacheTree* is created by associating each cache with a unique cache ID (*cid*) and mapping *cids* to their corresponding caches and parent caches (with 0 reserved for the root).

Caches are ordered by $\succ$, which determines which is more "recent" by comparing their

$$
\begin{aligned}
Op \triangleq\ & \texttt{pull} : \mathbb{N}_{nid} \to \Sigma \to \Sigma \\
\mid\ & \texttt{invoke} : \mathbb{N}_{nid} \to \mathit{Method} \to \Sigma \to \Sigma \\
\mid\ & \texttt{push} : \mathbb{N}_{nid} \to \Sigma \to \Sigma
\end{aligned}
$$

Figure 4.3: ADVERT operations.

$$
\begin{aligned}
C \uparrow C' &\triangleq C = \mathit{parent}(C') \vee C \uparrow \mathit{parent}(C') \\
C_1 \succ C_2 &\triangleq \mathit{time}(C_1) > \mathit{time}(C_2) \vee C_2 \uparrow C_1 \\
\mathit{freshCID}(tr) &\triangleq \max\{\mathit{cid}(C) \mid C \in tr\} + 1 \\
\mathit{addLeaf}(st, Q, C_P, C_{new}) &\triangleq \textbf{let } tr' = \mathit{tree}(st)[\mathit{freshCID}(\mathit{tree}(st)) \mapsto (C_P, C_{new})] \textbf{ in} \\
&\quad \textbf{let } act' = \mathit{active}[s \mapsto C_{new} \mid \forall s \in Q] \textbf{ in} \\
&\quad (tr', \mathit{times}(st), act') \\
\mathit{insertBtw}(st, Q, C_P, C_{new}) &\triangleq \textbf{let } tr = \mathit{tree}(st) \textbf{ in} \\
&\quad \textbf{let } tr' = tr[\mathit{cid}(C) \mapsto (\mathit{cid}(C_{new}), C_{new}) \mid \forall (\_, C) \in tr] \textbf{ in} \\
&\quad \textbf{let } act' = \mathit{active}[s \mapsto C_{new} \mid \forall s \in Q] \textbf{ in} \\
&\quad (tr'[\mathit{freshCID}(\mathit{tree}(st)) \mapsto (C_P, C_{new})], \mathit{times}(st), act') \\
\mathit{setTimes}(st, Q, t) &\triangleq (\mathit{tree}(st), \mathit{times}(st)[s \mapsto t \mid \forall s \in Q], \mathit{active}(st)) \\
\mathit{isLeader}(st, nid, C) &\triangleq \mathit{times}(st)[nid] = \mathit{time}(C) \wedge \mathit{caller}(C) = nid \\
\mathit{activeC}(tr, nid) &\triangleq \max_{\succ}\{C \in tr \mid \mathit{caller}(C) = nid \wedge C = \mathit{CCache}(\_)\} \\
\mathit{canElect}(st, C, Q) &\triangleq \forall s \in Q.\, C \succeq \mathit{active}(st)[s] \\
\mathit{canCommit}(C, nid, st) &\triangleq C = \mathit{MCache}(\_) \\
&\quad \wedge \mathit{isLeader}(st, nid, C) \wedge C \succ \mathit{activeC}(\mathit{tree}(st), nid)
\end{aligned}
$$

Figure 4.4: ADVERT auxiliary definitions.

timestamps. In the case of a tie, an ancestor is considered less recent than its descendants ($C \uparrow C'$ means C is an ancestor of C').

Operations ADVERT's interface for interacting with the cache tree consists of the `pull`, `invoke`, and `push` operations (Figure 4.3). The details of how exactly these operations are triggered is left up to the implementation and not exposed at the ADO level. For example, `invoke` could be the result of a client sending a message to the leader of a Paxos cluster, or it may be that the protocol is configured to periodically execute methods autonomously (e.g.,

PullOk

$$\frac{\mathbb{O}_{pull}(st, nid) = Ok(Q, Q_{ok}, C_{max}, t) \quad st' \triangleq setTimes(st, Q, t) \quad C_{new} \triangleq ECache(nid, t)}{\mathbb{O} \vdash \texttt{pull}(nid) : st \rightsquigarrow \textbf{if } Q_{ok} \textbf{ then } addLeaf(st', \{nid\}, C_{max}, C_{new}) \textbf{ else } st'}$$

InvokeOk

$$\frac{C_A \triangleq active(st)[nid] \quad isLeader(st, nid, C_A) \quad C_{new} \triangleq MCache(nid, time(C_A), M)}{\mathbb{O} \vdash \texttt{invoke}(nid, M) : st \rightsquigarrow addLeaf(st, \{nid\}, C_A, C_{new})}$$

PushOk

$$\frac{\mathbb{O}_{push}(st, nid) = Ok(Q, Q_{ok}, C_M) \quad st' \triangleq setTimes(st, Q, time(C_M)) \quad C_{new} \triangleq CCache(nid, time(C_M))}{\mathbb{O} \vdash \texttt{push}(nid) : st \rightsquigarrow \textbf{if } Q_{ok} \textbf{ then } insertBtw(st', Q, C_M, C_{new}) \textbf{ else } st'}$$

NoOp

$$\frac{}{\mathbb{O} \vdash op(nid) : st \rightsquigarrow st}$$

Figure 4.5: Semantics of Advert operations. Every operation may fail and have no effect on the state (either because an oracle returns *Fail* or the preconditions are not met), so NoOp 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.

Raft's heartbeat messages). In either case, Figure 4.5 defines the effect of each operation with the help of the auxiliary functions from Figure 4.4.

Pull Calling `pull` either successfully elects a leader and creates an *ECache*, fails to elect a leader but potentially still strips another replica's leadership, or has no effect. These outcomes are influenced by a variety of nondeterministic failures (e.g., dropped packets or crashed servers), but the precise cause is unimportant so we hide it behind an *oracle* ($\mathbb{O}_{pull}$). Oracles are deterministic, but network-based nondeterminism is modeled by quantifying over all valid oracles. Figure 4.6 defines the conditions that a valid oracle must satisfy.

If `pull` has any effect, the oracle returns a set of replicas (Q) that voted for the candidate, a cache (C_{max}), and a new timestamp (t). The timestamp must be strictly larger than the

$$\mathbb{O}_{pull} : \Sigma \to \mathbb{N}_{nid} \to (Ok(Set(\mathbb{N}_{nid}) * \mathbb{B} * Cache * \mathbb{N}_{time}) \mid Fail)$$
$$\mathbb{O}_{push} : \Sigma \to \mathbb{N}_{nid} \to (Ok(Set(\mathbb{N}_{nid}) * \mathbb{B} * Cache) \mid Fail)$$

ValidPullOracle
$$\frac{Q_{ok} \triangleq isQuorum(Q) \qquad \forall s \in Q.\, times(st)[s] < t \qquad canElect(st, C_{max}, Q)}{\mathbb{O}_{pull}(st, nid) = Ok(Q, Q_{ok}, C_{max}, t)}$$

ValidPushOracle
$$\frac{Q_{ok} \triangleq isQuorum(Q) \qquad \forall s \in Q.\, times(st)[s] \leq time(C_M) \qquad canCommit(C_M, nid, st)}{\mathbb{O}_{push}(st, nid) = Ok(Q, Q_{ok}, C_M)}$$

Figure 4.6: Valid `pull` and `push` oracle conditions.

current timestamps of the voters, and the cache must be at least as recent (with respect to $\succ$) as the voters' active caches (*canElect*). Depending on the size of Q, the timestamps, cache tree, and active caches are updated according to Figure 4.5.

If the voters form a quorum (Q_{ok}), their timestamps are updated to reflect their votes (*setTimes*), an *ECache* is added to C_{max}, and the leader's active cache is updated to the *ECache*. If there are not enough voters, then no cache is created, but the timestamps are still updated, which may be enough to block another replica from committing its methods. It is also possible for the oracle to return *Fail*, in which case no change is made.

Invoke Invoking a method requires that the caller has already successfully called `pull`. It confirms this by checking that its current timestamp matches the timestamp of its active cache and that the active cache was created by itself (*isLeader*), which is only possible as leader. This check would fail if the replica either was never the leader, or if it was elected but has since voted for a more recent leader. Otherwise, it creates a new *MCache* and adds it to the end of its active branch.

Push As with `pull`, `push` succeeds or fails based on the outcome of $\mathbb{O}_{push}$. Recall that when a client calls `push` after a sequence of method invocations, some suffix of *MCaches* may not be committed. $\mathbb{O}_{push}$ allows this by choosing an arbitrary cache (C_M) that satisfies *canCommit*. This requires that C_M is an *MCache*, created by the caller in the current round, that is more recent than the caller's latest *CCache* (*activeC*). The voter's timestamps must also be no larger than C_M's, though they may be equal.

If `push` receives a quorum of votes, it updates the voters' timestamps and creates a *CCache*. However, rather than adding the new cache as a leaf, it is inserted between C_M and its descendants (*insertBtw*). This marks C_M and its ancestors as committed, while still leaving open the possibility for the remaining *MCaches* to be committed by a later `push`. As with `pull`, a non-quorum set of voters causes only the timestamps to change, and a *Fail* result has no effect.

4.3 Single-ADO Reasoning

This section defines several important properties of the ADO model and highlights some of the differences between distributed and sequential or concurrent objects. For ease of presentation, example ADOs use an object-oriented pseudocode rather than the formal representation from Figure 4.2. For example, in the simple `BankAccount` ADO (Figure 4.7), the line beginning with **`shared`** `balance` indicates that the replicated data has type $\mathbb{Z}$ and is initialized to 0. Lines beginning with **`method`** define the method interface. Method bodies are written in an imperative style and use `this` to access the current state (the comments show the equivalent functional versions). When calling `pull`, `invoke`, or `push`, the caller's

```
ADO BankAccount {
  shared balance : ℤ := 0; /* Σ = ℤ */
  /* read() ↦ λ bal. (bal, bal) */
  method read() { return this.balance; }
  /* deposit(n) ↦ λ bal. (bal + n, tt) */
  method deposit(n) { this.balance += n; }
  /* withdraw(n) ↦ λ bal.
  *   if n ≤ bal then (bal - n, n) else (bal, 0) */
  method withdraw(n) {
    if (n ≤ this.balance) {
      this.balance -= n;
      return n;
    } else { return 0; }
  }
}
```

Figure 4.7: Distributed bank account object.

node ID is passed implicitly and is accessible from within the method body as `this.nid`.

4.3.1 Programming with ADOs

One major difference between ADOs and sequential or concurrent objects is that, although individual ADO operations are atomic, calling a method (preparing the object with `pull`, invoking the method, and committing the result with `push`) is three separate steps, which may interleave with each other. Each step can also fail, and depending on how the failures are handled the method call can exhibit different behaviors. The most common behaviors are at-most-once, at-least-once, and exactly-once [Felber et al. 2001; Ramalingam and Vaswani 2013]. In SMR-based interfaces these different options are typically hidden behind a single, black-box RPC operation [Burrows 2006; Schneider 1990; Wollrath et al. 1996], but the ADO model offers the flexibility to precisely specify which one an application should use depending on the situation.

At-Most-Once As the name suggests, a method called with at-most-once semantics is guaranteed to be applied to the object either once, or not at all.

Definition 1 (Called At-Most-Once)**.** *A method M is called at-most-once by nid if there exists no more than one MCache in the cache tree containing M that belongs to nid.*

This behavior can be useful when a method's side effects should not execute twice, and the application is able to tolerate unclear outcomes (i.e., a sort of speculative execution). Note that this definition only allows a method to be called at most once *ever*. However, since arguments are part of the method name (e.g., `m(1)` and `m(2)` are considered different methods), a simple solution is to add a "request ID" argument to each method and use a fresh ID on every call.

A method can be called with at-most-once semantics by calling `pull`, invoking the method, calling `push`, and aborting if any step fails. We abbreviate this sequence of operations as `obj.m()?`. In the following pseudocode `pull` returns either the state corresponding to the chosen cache or the special value **`FAIL`**. Similarly, `push` returns either the return value of the last committed method or **`FAIL`**. Formally, these are functions on Σ, but for simplicity we use a more concise imperative notation in which the state is implicitly threaded through each operation.

```
obj.m()? := if (obj.pull() != FAIL) { obj.invoke(m); return obj.push(); }
            else { return FAIL; }
```

At-Least-Once When an application cannot tolerate a failed method call, the obvious solution is to retry it. Note, however, that "failed" in a distributed setting just means "not definitely successful", and an uncommitted *MCache* might be committed by a later `push`. Thus, retrying a method until it succeeds only guarantees that it is called *at least* once.

Definition 2 (Called At-Least-Once)**.** *A method M is called at-least-once by nid if there exists an MCache in the cache tree containing M that belongs to nid.*

This is appropriate for read-only methods (e.g., `read`), or if a method's effect on the internal state is effectively invisible to an outside observer (e.g., a random number generator changing its seed). To make an at-least-once call (`obj.m()`$^+$), one can repeat an at-most-once call `obj.m()`? with a fresh request ID until it succeeds.[1]

```
obj.m()⁺ := do {
              rqID := /* compute fresh ID */;
              ret := obj.m(rqID)?;
            } while (ret = FAIL);
            return ret;
```

Exactly-Once Often the most intuitive behavior is for a method to execute precisely once. The `withdraw` method, for example, is difficult to use sensibly without exactly-once semantics. In general, however, this is impossible to achieve because invoking a method only once does not guarantee that it will be committed, but invoking it more than once may commit it multiple times. If, however, a method is idempotent (i.e., applying it more than once has the same effect as applying it once), repeatedly calling it with at-most-once semantics will produce an equivalent result to an exactly-once behavior (`obj.m()!`). The method may in fact be committed several times, but everything after the first commit has no effect, so it is as if it was committed once.

Definition 3 (Called Exactly-Once)**.** *A method M is called exactly-once by nid if there exists an MCache in the cache tree containing M that belongs to nid, and, for all states st, $M(M(st)) = M(st)$.*

[1] This assumes the loop eventually terminates. We defer liveness considerations to Chapter 6.

```
obj.m()! := do { ret := obj.m(rqID)?; } while (ret = FAIL); return ret;
```

The idempotency requirement is less restrictive than it might seem because any method can be mechanically transformed into an idempotent one by simply caching the result using the node and request IDs as keys. This ensures that the method's side effects only execute once and subsequent calls return the memoized value. For example, the `withdraw` method could be transformed as follows.

```
/* Add results : Map[(ℤ * ℤ) ⇀ ℤ] to the replicated state */
method withdraw(rqID, n) {
  if ((this.nid, rqID) ∈ this.results) { /* Already executed, do nothing */
  } else if (n ≤ this.balance) {
    this.balance -= n;                   /* Only change the balance once */
    this.results[(this.nid, rqID)] := n; /* Store the return value */
  } else {
    this.results[(this.nid, rqID)] := 0;
  }
  return this.results[(this.nid, rqID)]; /* Return the cached result */
}
```

4.3.2 Proving with ADOs

Reasoning about an ADO can be quite straightforward because the object's behaviors are fully captured by log of atomic events. As an example, we sketch a proof that the balance of a `BankAccount` object is always non-negative.

Proof Sketch. We proceed by induction on a sequence of ADO operations. In the base case the account balance has its initial value of 0, which is non-negative. Now suppose we have an arbitrary `BankAccount` object with a non-negative balance and consider every possible next step. Neither `pull` nor `invoke` can change the committed state so the balance remains non-negative. In the case of a successful `push`, some sequence of methods may be committed. Of the possible methods, `withdraw` is the only one that could cause the

balance to decrease; however, it ensures that the amount to be deducted is not greater than the current balance. Therefore, for any sequence of ADO operations, the balance can never be negative. □

`BankAccount` in Figure 4.7 is only a specification and must be implemented by a protocol such as Paxos or Raft in order to run. The choice of protocol is critical for achieving good performance, but a significant strength of the ADO model's unifying specification language is that one can make this decision orthogonally from correctness considerations. As long as the protocol satisfies the ADO semantics, one can reuse the same specification and application-level proofs.

4.4 ADO Composition

The ability to easily compose components is critical for building scalable, reliable distributed applications. It allows complex systems to be decomposed into modular pieces that are easier to understand and to fine-tune performance. Compositional reasoning is simpler with Advert than with network-based specifications because an application's behavior can be understood purely in terms of `pull`, `push`, and its methods without any knowledge of the underlying distributed protocol. Despite its simplicity, this interface is more expressive than SMR because it offers more control over failure handling (e.g., the different method call semantics in Section 4.3.1).

ADOs are internally implemented by a cluster of servers that only communicate amongst themselves and are not aware of other ADOs. Therefore, composing two ADOs really means composing their interactions with clients. For example, a client of ADOs `A`

and B might execute `x := A.a()!; B.b(x)!`, thus creating a composite system involving the client and both objects, which we refer to as a *distributed application*, or DApp. It is impossible to prove much if clients are allowed to interact with objects arbitrarily, therefore DApps limit client behaviors to a set of predefined *procedures*. For example, a simple DApp composing two `BankAccounts` (Figure 4.7) to allow transfers between them could be expressed as follows.

```
DApp TransferAccount(acct1: BankAccount, acct2: BankAccount) {
  proc transfer1to2(n) {
    if (acct1.withdraw(n)! = n) { acct2.deposit(n)!; }
  }
  proc transfer2to1(n) {
    if (acct2.withdraw(n)! = n) { acct1.deposit(n)!; }
  }
}
```

Unlike ADO methods, DApp procedures are not necessarily atomic and it is entirely possible for concurrent executions of `transfer1to2` and `transfer2to1` to interleave. If, however, one can prove that a particular DApp's procedures are atomic, then the composite system logically behaves as if it were implemented by a single consensus protocol, and therefore has an equivalent ADO specification.

4.4.1 Case-Study: Key-Value Stores

To demonstrate ADO composition in action, we present three versions of a key-value store: a self-contained ADO, a lock-based DApp, and, most interestingly, a lock-free DApp. Each store maps the hash of a key to a value along with metadata about the value (e.g., its size in memory). For simplicity, we do not consider liveness or hash collisions; nevertheless, the data structures and coordination patterns in these examples are similar to those employed

```
ADO KVPrimitive {
  shared kv : Vector[ℤ * ℤ]; /* (meta, value) */
  method set(k, v) { this.kv[hash(k)] := (sizeof(v), v); }
  method lookup(k) { (_, v) := this.kv[hash(k)]; return v; }
  method getmeta(k) { (m, _) := this.kv[hash(k)]; return m; } }
```

Figure 4.8: Single ADO key-value store.

in real systems [Burrows 2006; Chang et al. 2006] and could scale to larger applications.

The first example, `KVPrimitive` (Figure 4.8), is an ADO that manages both the data and metadata. This has the advantage of making the specification quite simple, and it guarantees for free that the data and metadata are updated atomically will remain synchronized.

Lock-Based Version `KVPrimitive`'s simplicity comes at the cost of some control over performance and reliability. For example, the data and metadata cannot be stored on separate clusters of servers, which might be desirable to reduce the risk of losing both to replica crashes. `KVLock` enables this type of implementation choice by composing separate `DVector` objects (an ADO wrapper around a sequential `Vector`) for the data and metadata (Figure 4.9). Because these are now independent objects, keeping them synchronized also requires composing them with a distributed lock object (`CASLock`).

For `KVLock` to implement `KVPrimitive`, its procedures must behave atomically. Every procedure is protected by a lock, so this is trivial as long as `CASLock` guarantees mutual exclusion. Although we use `CASLock` for simplicity, one can also design more sophisticated ADO locks (see Appendix A.1).

Lemma 1 (`CASLock` Mutual Exclusion)**.** *If, at some point, the lock is held by nid, and then at a later time it is held by a different nid′, there must have been a committed* `release` *method called by nid in between.*

```
ADO DVector[T] {
  shared data : Vector[T] := [];
  ... /* insert, append, pop, etc. */
}
```

```
ADO CASLock {
  shared owner : option ℕ := None;
  method tryAcquire() {
    if (this.owner = None) {
      this.owner := Some(this.nid);
    }
    return this.owner = Some(this.nid);
  }
  method release() {
    if (this.owner = Some(this.nid)) {
      this.owner := None;
    }
  }
}
```

```
DApp KVLock(lk: CASLock, data: DVector[ℤ], meta: DVector[ℤ]) {
  proc set(k, v) {
    while (!this.lk.tryAcquire()!) {}
    this.meta.insert(hash(k), sizeof(v))!;
    this.data.insert(hash(k), v)!;
    this.lk.release()!;
  }
  proc lookup(k) {
    while (!this.lk.tryAcquire()!) {}
    v := this.data.get(hash(k))!;
    this.lk.release()!;
    return v;
  }
  proc getmeta(k) {
    while (!this.lk.tryAcquire()!) {}
    m := this.meta.get(hash(k))!;
    this.lk.release()!;
    return m;
  }
}
```

Figure 4.9: Lock-based composite ADO key-value store.

Proof Sketch. We proceed by induction on a sequence of ADO operations between the point where *nid* holds the lock, and where *nid′* holds it. If there are no operations, then *nid* = *nid′*, which contradicts the assumption that they are different. Otherwise, some sequence of methods must have been committed. If any of the methods is `release(nid)`, we are done. If not, the only other options are `tryAcquire`, or `release` for some other client. If we can show that neither option changes the lock owner, then we are done by the inductive hypothesis.

For `tryAcquire` we know that `this.owner = Some(nid)`, which is not `None`, so the method will not change the owner. Similarly, for `release`, we know the caller is not the current lock owner, so `this.owner` does not equal `Some(this.nid)` and the method has no effect on the owner. □

The next step is to show that `KVLock` simulates `KVPrimitive`. This involves defining a relation between their states and proving that, for each of `KVPrimitive`'s methods, there is a `KVLock` procedure that preserves the relation. We only show the case for the `set` method as the cases for `lookup` and `getmeta` are similar.

Lemma 2 (`KVLock`-`KVPrimitive` Refinement)**.** *Suppose a* `KVLock` *and* `KVPrimitive` *are related such that, for every key, both objects store the same value and metadata. If both successfully commit the* `set` *method, then this relation is preserved.*

Proof Sketch. Let the newly inserted key-value pair be called *k* and *v*. It is enough to show that, after `set`, only the value corresponding to *k* is changed, and that, for both objects, the new value is *v*. This is trivial for `KVPrimitive` as `set` atomically updates `this.kv`.

For `KVLock`, it first waits to acquire a lock using an exactly-once method call (Line 3),

```
DApp KVLockFree(data: DVector[ℤ], meta: DVector[ℤ * ℤ]) {
  proc set(k, v) {
    idx := this.data.append(v)!;
    this.meta.insert(hash(k), (sizeof(v), idx))!;
  }
  proc lookup(k) {
    (_, idx) := this.meta.get(hash(k))!;
    return this.data.get(idx)!;
  }
  proc getmeta(k) { (m, _) := this.meta.get(hash(k))!; return m; }
}
```

Figure 4.10: Lock-free composite key-value store.

which, for simplicity, we assume eventually succeeds.[2] At this point it has exclusive access to `this.meta` and `this.data`, so the following `insert` method calls execute atomically. Now both vectors have been updated at k and nowhere else, so the relation with `KVPrimitive` is preserved. □

Lock-Free Version A lock is a simple solution for synchronizing distributed components, but it is often a performance bottleneck, and can cause deadlock if the owner dies while holding it. Therefore, a lock-free solution such as `KVLockFree` (Figure 4.10) may be preferable. Like `KVLock` it delegates data and metadata storage to `DVector` objects, but instead of synchronizing them with a lock, it relies on the order in which `data` and `meta` are updated. In `set`, the value is appended to `data`, which returns an index pointing to the end of the `Vector`. The key is then mapped to this index and the value's size in `meta`. To recover the value, `lookup` follows the reverse order by first reading the index from `meta` and using it to access `data`.

The lack of mutual exclusion makes the atomicity of these procedures less obvious than for `KVLock`. The key observations are that `data.append()` returns a monotonically

[2] In reality, a system would set some upper bound on the number of attempts to avoid blocking forever.

increasing index equal to the length of `data` before the append, and that `meta.insert()` is the linearization point (i.e., the moment when a new key-value mapping is visible to a client). We sketch the case where `set` and `lookup` are executed concurrently with the same key (`k`) by threads `T1` and `T2` respectively. Two possible interleavings of these procedures are expressed below, and the other cases are similar.

```
T2: (_, idx1) := meta.get(hash(k));
T1: idx2 := data.append(v);
T2: data.get(idx1);
T1: meta.insert(hash(k),
      (sizeof(v), idx2));
```

or

```
T1: idx1 := data.append(v);
T2: (_, idx2) := meta.get(hash(k));
T1: meta.insert(hash(k),
      (sizeof(v), idx1));
T2: data.get(idx2);
```

Proof Sketch. At the beginning of each case we have the invariant that the length of the `data` vector is greater than every index stored in the `meta` vector. This can be seen by observing that only `set` modifies `data` or `meta` and the index it inserts is equal to `data.len - 1`. As `data` only ever becomes longer, this means the indices always point to a valid position in the vector.

Therefore, we have `idx1` < `idx2` in the left case and `idx2` < `idx1` in the right. This means `data.get` and `data.append` touch disjoint entries in the `Vector`, so they can safely commute without changing the result (swap Lines 2 and 3 in the left case). In the right case, however, the operations on `meta` are between the `data` operations. Since `meta` and `data` are separate objects, their methods may also commute as long as program order is preserved (swap Lines 1 and 2 & Lines 3 and 5). After reordering, both cases are exactly equivalent to atomically executing `lookup(k)` followed by `set(k, v)`, which proves that the methods are linearizable. □

Lock-free systems can be much more performant than lock-based ones, but to our knowledge no other verification framework has handled an example like `KVLockFree`.

4.4.2 Alternate Method-Calling Patterns

Exactly-once method calls are intuitive, but alternate method-calling patterns can sometimes improve performance by sending fewer messages. This involves rewiring method calls and handling failures at a lower level of abstraction than is typically available in SMR-like models. With Advert, by simply adjusting when `pull` and `push` are called, one can express and reason about a variety of optimized and unoptimized versions of an application using an atomic interface that is both simpler than a network model and more general because it is not tied to a specific protocol. A real-world example of this type of optimization is in Two-Phase Commit (2PC) combined with consensus (e.g., Paxos Commit [Gray and Lamport 2006] and WormTX [Shin et al. 2019]).

The standard 2PC protocol distributes its state across a set of resource managers (RM), each of which stores a list of operations to apply. To ensure consistency among the RMs, a transaction manager (TM) first asks each if it can apply an operation locally. If all vote yes, then the TM tells them to commit and apply the operation, and otherwise it tells them to abort. This all-or-nothing behavior means the entire system blocks if a single RM becomes unresponsive. Replicating each RM with a consensus protocol reduces this risk by allowing them to survive f crashes out of $2f + 1$ servers.

Figure 4.11 shows a simple implementation of this version of 2PC using n ADOs to model the replicated RMs. For simplicity, we assume the TM never crashes and handles one request at a time. A more realistic version that properly handles state recovery after a TM crash can be found in Appendix A.2. The code is mostly unsurprising, but there are two points that deserve attention.

```
DECISION := YES | NO | COMMIT | ABORT;
TX := {ops: Vector[IO]; ts: ℤ; decision: DECISION};
/* Local decision to vote YES or NO if tx can be applied to txs */
func tx_can_commit(txs, tx) { ... }
ADO RM {
  shared txs : Vector[TX] := [];
  method prepare(tx) {
    tx.decision := tx_can_commit(this.txs, tx);
    this.txs.append(tx);
    return tx.decision;
  }
  method decide(ts, decision) {
    idx := this.txs.find(λ tx. tx.ts = ts);
    this.txs[idx].decision := decision;
  }
}
```

```
DApp TM(rm_1: RM, ..., rm_n: RM) {
  local ts : ℤ := 0;
  /* Must be called once when TM starts */
  proc init() {
    for rm in [this.rm_1, ..., this.rm_n] {
      while (rm.pull() = FAIL) {}
    }
  }
  proc handle_request(ops) {
    this.ts += 1;
    tx := {ops=ops, ts=this.ts, decision=COMMIT};
    /* Phase 1: Collect decisions */
    for rm in [this.rm_1, ..., this.rm_n] {
      /* Method invocation only, not an exactly-once call */
      rm.invoke(prepare(tx));
      for i in 0..MAX_TRY {
        res := rm.push();
        if (res != FAIL) { break; }
      }
      /* Abort and break if RM says no or can't commit in MAX_TRY tries */
      if (res = NO || res = FAIL) {
        tx.decision := ABORT;
        break;
      }
    }
    /* Phase 2: Commit the decision */
    for rm in [this.rm_1, ..., this.rm_n] {
      rm.invoke(decide(tx.ts, tx.decision));
      while (rm.push() = FAIL) {}
    }
  }
}
```

Figure 4.11: Two-Phase Commit with replicated RMs.

```
1  ADO Transaction {
2    shared ts : ℤ := 0;
3    shared txs_1 : Vector[TX] := []; ...; shared txs_n : Vector[TX] := [];
4    method handle_request(ops) {
5      this.ts += 1;
6      tx := {ops=ops, ts=this.ts, decision=COMMIT};
7      /* Phase 1: Collect decisions and abort if any vote no */
8      if ([this.txs_1, ..., this.txs_n].any(λ txs. tx_can_commit(txs, tx) = NO)) {
9        return;
10     }
11     /* Phase 2: Commit the decision */
12     for txs in [this.txs_1, ..., this.txs_n] {
13       txs.append(tx);
14     }
15   }
16 }
```

Figure 4.12: Transaction ADO.

The first is the `init` procedure, which simply calls `pull` on every RM. This only needs to be called once when the TM starts because 2PC assumes that there is at most one valid TM that can issue transactions to the RMs, so there is no chance for another leader to be elected. This means that, unlike exactly-once calls, the method calls on Lines 15 and 28 can skip `pull` and proceed straight to `invoke`. A similar optimization is also possible for `KVLock` because only the client that holds the lock can modify `data` and `meta`, so there is no risk of preemption.

The second place that Figure 4.11 differs from the previous examples is Line 16 in `handle_request`. Unlike exactly-once calls, which retry `push` infinitely, this sets an upper bound on the number of failed attempts. If this limit is reached, the TM safely treats it as a `NO` vote and aborts the operation. This fine-grained control over failure handling is one way Advert facilitates optimized system designs.

As in the previous examples, we can show that this DApp refines an ADO specification (Figure 4.12). Because we assume there is only one client at a time, we can consider the

procedures atomic. Then the `Transaction` ADO is nearly the same as the `TM` DApp with inlined RMs, so it is easy to see how they relate. One minor difference is that, in phase 1, `TM` treats an unresponsive RM as a `NO` vote, so a transaction may be allowed according to `tx_can_commit`, but `TM` aborts it anyway. This cannot happen in `Transaction` because there are no RMs to be unresponsive. Nevertheless, one can prove a soundness relation that says the DApp commits a transaction only if the ADO does as well.

4.5 Refinement

Thus far, we have seen how ADOs and DApps can be reasoned about independently from their protocol-level implementations. This section discusses how the lower-level implementations relate to their ADO specifications.

4.5.1 Network-Based Specifications

The gap between the ADO model and C code (atomic methods and a logical cache tree vs. packets and concrete memory) is too large to cover in a single step. To help close it, we introduce an intermediate "network-based" specification that more closely matches the implementation but still abstracts away C-specific details. One can then link the specifications with contextual refinement [Gu et al. 2015; Liang et al. 2013] to achieve an end-to-end correctness property.

State The global state of the system is modeled as a set of local replica states (*Replica*) and a *Network*, which is a pair of bags of sent and delivered messages (Figure 4.13). A

$$
\begin{aligned}
\Sigma_{\text{net}} &\triangleq (\mathbb{N}_{nid} \rightharpoonup \mathit{Replica}) * \mathit{Network} \\
\mathit{Replica} &\triangleq \mathbb{N}_{time} * \mathit{rdata} \\
\mathit{Network} &\triangleq \mathit{Set}(\mathit{Msg}) * \mathit{Set}(\mathit{Msg}) \\
\mathit{Msg} &\triangleq \mathit{Request}(\mathbb{N}_{nid} * \mathbb{N}_{nid} * \mathbb{N}_{time} * \mathit{Cmd}) \\
&\quad \mid \mathit{Ack}(\mathbb{N}_{nid} * \mathbb{N}_{nid} * \mathbb{N}_{time} * \mathit{Cmd}) \\
&\quad \mid \mathit{Ghost}(\mathbb{N}_{nid} * \mathbb{N}_{time} * \mathit{GCmd}) \\
\mathit{Cmd} &\triangleq \mathit{Elect} \mid \mathit{Commit}(\mathit{rdata}) \\
\mathit{GCmd} &\triangleq \mathit{Begin}(\mathit{Cmd}) \mid \mathit{End}(\mathit{Cmd}, \mathbb{B}) \\
\mathit{Op}_{\text{net}} &\triangleq \texttt{elect} : \mathbb{N}_{nid} \to \Sigma_{\text{net}} \to \Sigma_{\text{net}} \\
&\quad \mid \texttt{invoke} : \mathbb{N}_{nid} \to \mathit{Method} \to \Sigma_{\text{net}} \to \Sigma_{\text{net}} \\
&\quad \mid \texttt{commit} : \mathbb{N}_{nid} \to \Sigma_{\text{net}} \to \Sigma_{\text{net}} \\
&\quad \mid \texttt{deliver} : \mathit{Msg} \to \Sigma_{\text{net}} \to \Sigma_{\text{net}}
\end{aligned}
$$

Figure 4.13: Generalized Paxos network-based state and operations.

$$
\begin{aligned}
&\mathit{rdata} : \mathit{Type} && \mathit{update} : \mathit{Method} \to \mathbb{N}_{time} \to \mathit{rdata} \to \mathit{rdata} \\
&\mathit{rtime} : \mathit{rdata} \to \mathbb{N}_{time} && \mathit{isQuorum} : \mathit{Set}(\mathbb{N}_{nid}) \to \mathbb{N}_{time} \to \mathit{Cmd} \to \mathbb{B}
\end{aligned}
$$

Figure 4.14: Generalized Paxos parameters. The type of the replicated state (*rdata*) remembers the time it was created (*rtime*) and can be modified with an update function (*update*). Quorums are decided by *isQuorum*, which may depend on the current command and timestamp in addition to the set of replicas.

message (*Msg*) may be a requests (*Request*), which contains a sender, recipient, logical time, and command, a request acknowledgement (*Ack*), or a logical *Ghost* event. A command is either a candidate's request to be elected (*Elect*) or an attempt to commit a new state (*Commit*). Ghost events (*GCmd*) on the other hand do not represent real communications, but are logical markers of the start and end of an election or commit attempt. The *End* marker indicates whether the leader successfully collected a quorum of votes.

A replica's local state consists of the current largest timestamp it has observed and its snapshot of the replicated state, which is represented by the parameterized *rdata* type. This, along with a few other parameters (Figure 4.14), generalizes the specification to

describe a larger class of protocols. For example, *rdata* could either be instantiated to a log of commands for Multi Paxos or just a single value for Single Paxos. In order to compare two *rdatas* to determine which is more recent, they must store the time at which they were committed (*rtime*). There must also be an *update function* (*update*) to modify an *rdata* when a *Method* is committed. Finally, we generalize the definition of a quorum (*isQuorum*) to support more than just the standard simple majority. Section 4.5.3 provides instantiations of these parameters for four Paxos variants.

Operations Network-level behaviors are defined by the `elect`, `invoke`, `commit`, and `deliver` operations. The first three represent a candidate or leader beginning an operation by performing some local state updates and sending a request and a ghost *Begin* event. For example, `elect` increments the local timestamp and broadcasts an *Elect* request, `invoke` locally applies the update function to its *rdata*, and `commit` broadcasts the new *rdata* in a *Commit* request (Figure 4.15).

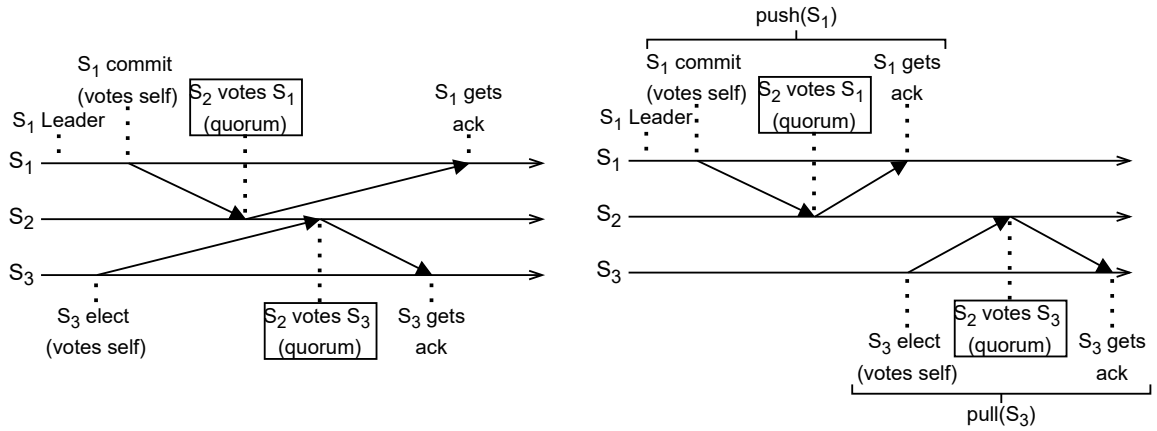
The `deliver` operation can be triggered at any time by an oracle, which arbitrarily chooses a sent message to arrive at its intended recipient. Messages may be delivered multiple times or not at all, but the contents are never corrupted. Upon receiving a message the appropriate handler is triggered (Figure 4.16), which decides whether to vote for or ignore the request, and, in the former case, updates the recipient's local state and sends an acknowledgement. Leaders keep track of their acknowledgements and emit an *End* ghost event once either they have a quorum (according to *isQuorum*) or they abandon the request after being preempted.

```
send(st, msg) ≜
  let sent′ = sent(network(st)) ∪ {msg} in setNetwork((sent′, recvd(network(st))))
broadcast(st, msgs) ≜ fold(send, msgs, st)
elect(nid, st) ≜
  let st′ = incTime(nid, st) in
  let ghost = Ghost(nid, time(nid, st), Begin(Elect)) in
  let reqs = {Request(nid, recip, time(st′, nid), Elect) | ∀recip ∈ replicas} in
  broadcast(send(st′, ghost), reqs)
invoke(nid, m, st) ≜
  let data′ = update(m, time(nid, st), data(nid, st)) in setData(st, nid, data′)
commit(nid, st) ≜
  let t = time(st, nid) in
  let data = data(st, nid) in
  let ghost = Ghost(nid, t, Begin(Commit(data))) in
  let reqs = {Request(nid, recip, t, Commit(data)) | ∀recip ∈ replicas} in
  broadcast(send(st, ghost), reqs)
```

Figure 4.15: The `elect`, `invoke`, and `commit` network state transition functions.

```
canCommit(st, nid, t, data) ≜ t ≥ time(st, nid) ∧ rtime(data) > rtime(data(st, nid))
canCommitAck(st, nid, t) ≜ t = time(st, nid)
handleCommit(st, nid, t, data) ≜ setData(setTime(st, nid, t), nid, data)
deliver(msg, st) ≜
  if msg = Request(from, to, t, Commit(data)) ∧ canCommit(st, to, t, data) then
    let st′ = handleCommit(to, t, data) in
    let ack = Ack(to, from, t, Commit(data)) in
    send(st′, ack)
  else if msg = Ack(from, to, t, Commit(data)) ∧ canCommitAck(st, to, t) then
    let st′ = addAck(to, msg) in
    let ghost = Ghost(to, t, End(Commit(data), true)) in
    if isQuorum(acks(st, to)) then send(st′, ghost) else st′
  else ...
```

Figure 4.16: Selected `deliver` request handlers.



(a) Interleaved `elect` and `commit` messages. S_3 begins the election first, but S_1's commit reaches a quorum of voters first, though the acknowledgements arrive later.

(b) The same `elect` and `commit` messages, but sorted to reflect their equivalent linearized order.

Figure 4.17: Linearizing asynchronous network events.

4.5.2 Relating Network and ADO Models

We show that Advert correctly captures the behaviors of the network-based specification by proving a refinement between `pull` and `elect`, and likewise for `push` and `commit`. These theorems state that matching Advert and network-based states continue to match after taking a step.

What it means for the states to "match" is defined by the refinement relation $\mathbb{R}$. Intuitively, it holds when the replicated state in both models is observably equivalent. This roughly means the committed methods in every replica's local log all have corresponding *CCaches* in the cache tree. See Appendix B.1 for more precise definitions.

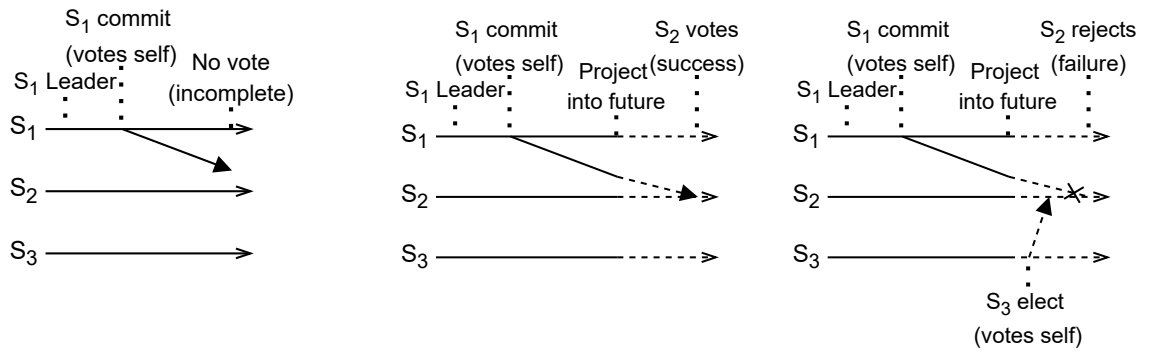
The challenge is that the asynchronous network sometimes creates situations that do not line up cleanly with a sequence of ADO events. To resolve these mismatches, we must transform the trace of network events into an equivalent one that is more suitable.

Reordering the Network In Figure 4.17a, the current leader S_1 tries to commit a method just after S_3 begins an election and their messages interleave. Each replica succeeds by receiving a quorum of votes (from itself and S_2), but to map these messages to Advert operations, we must determine which completed first. Although S_3 begins first and receives acknowledgments first, what actually decides when an operation takes effect is when it is received by a quorum of replicas, which in this case is determined by the order that S_2 observes the requests. Therefore, we may logically reorder the messages as in Figure 4.17b to create an equivalent linearized timeline whose corresponding Advert events are S_1's `push`, followed by S_3's `pull`.

To see why this is the correct ordering, consider the alternative. In the interleaved timeline, S_2 accepts S_1's commit request before voting for S_3, which means its acknowledgment will include S_1's newly committed command. This fits with the interpretation that `push` comes before `pull`, because then $\mathbb{O}_{pull}$ is allowed to select S_1's *CCache*, so we maintain the relation between S_3's network-level local log and its active cache. If instead `pull` came first, it would be impossible for S_3 to observe S_1's new method and the relation would be broken.

Completing the Network Figure 4.18a illustrates another problem: network operations are non-atomic and may be in an incomplete state. Here, for example, S_1's commit request has received only one vote (its own). It is certainly not successful, but it not correct to consider it a failed `push` yet either because it might still gather enough votes.

In order to disambiguate these cases and assign this operation a corresponding Advert operation, we introduce a *phase scheduler*, which is an oracle that, given a trace of network



(a) An incomplete commit request with only one vote.

(b) A potential future in which the commit succeeds by gaining S_2's vote.

(c) A different future in which the commit fails from being preempted by S_3's election.

Figure 4.18: Completing network events.

events, nondeterministically decides the next event. By repeatedly querying the phase scheduler, we can *complete* the network by extending the trace arbitrarily far into the future until there is enough information to decide an operation's fate.

For example, Figure 4.18b shows a future in which S_2 receives and acknowledges the request, so the `push` is successful. Figure 4.18c shows an alternative where S_3 begins an election, which causes S_2 to reject the commit request. Because both S_3 and S_2 have moved on to a larger timestamp, S_1's request can never receive a quorum of votes so it has definitely failed. This approach is related to a similar notion of completion from Izraelevitz et al. [2016] who used it to prove linearizability of shared-memory concurrent systems in the presence of crashes.

Matching Events By combining these operations, one can transform a trace of potentially incomplete, interleaved network events into an equivalent trace in which events are untangled, grouped together, and have clear beginnings and endings. This makes it simple to map them to their corresponding atomic ADVERT events. However, for the relation

defined by this mapping to be meaningful, it must guarantee that matching event traces produce matching states; i.e., a committed method in the cache tree should also appear in a quorum of replicas' local logs. To prove this we must show that the network specification satisfies replicated state safety.

4.5.3 Safety Proof Template

Proving safety is the most challenging step of the refinement because it requires reasoning at the network level and working with concurrent, non-atomic events. In Chapters 5 and 6 we will see that it is possible to avoid this step and instead prove safety at the ADO level by storing additional metadata about quorums of voters in the caches. Advert lacks this information, but fortunately, the proof must only be done once per protocol. Furthermore, many distributed protocols are minor variations of the same concept that all rely on the same core safety argument. Paxos, for example, has many variants (e.g., Fast Paxos [Lamport 2006], Disk Paxos [Gafni and Lamport 2003]), but their correctness always relies on consecutive `elect` and `commit` phases having overlapping quorums of supporters, which prevents different commands from being committed in the same slot.

Our network-based specification for Paxos takes advantage of these similarities by parametrizing certain protocol-specific details, which can be instantiated to accommodate a range of Paxos variants. With some basic assumptions (e.g., quorums have a non-empty intersection), we can prove safety once and for all in terms of these parameters, which creates a reusable *proof template* for safety that holds for a family of Paxos-like protocols. To instantiate the template, one simply needs to define the parameters and prove that they

satisfy the assumptions, after which the top-level theorem is derived for free.

Besides the generalized parameters, the proof mostly follows a standard approach [Lamport 2001; van Renesse and Altinbuken 2015] and has little to do with the ADO model, so its details are omitted. More information about the high-level proof structure can be found in Appendix C.1. The following are a few sample instantiations of the parameters, which include the type of the replicated state (*rdata*), a function to determine if a set of replicas constitutes a quorum (*isQuorum*), and an update function (*update*), which is responsible for computing a new *rdata* when a method is committed. Each of these definitions is formalized and proved to satisfy the necessary assumptions in Coq, but here we present them in a more readable notation.

Single Paxos [Lamport 2001] replicates a single, immutable value. This is typically some state machine command, which may be serialized as a string or an integer. The update function enforces immutability by only accepting the new value if the old state has not already been set (represented by $\perp$). Quorums are decided by a simple majority ($f + 1$ assuming a set of $2f + 1$ replicas).

$$
\begin{aligned}
rdata &\triangleq (\mathbb{Z} * \mathbb{N}_{time}) \mid \perp \\
rtime(data) &\triangleq \textbf{if } data = (\_, t) \textbf{ then } t \textbf{ else } 0 \\
update(m, t, data) &\triangleq \textbf{if } data = \perp \textbf{ then } (m(), t) \textbf{ else } data \\
isQuorum(votes, t, cmd) &\triangleq |votes| \geq f + 1
\end{aligned}
$$

Multi Paxos [van Renesse and Altinbuken 2015] extends Single Paxos to a log of immutable values. Quorums are the same as in Paxos, but the update function now appends a new entry to the end of the log.

$$
\begin{aligned}
rdata &\triangleq List(\mathbb{Z} * \mathbb{N}_{time}) \\
rtime(data) &\triangleq \textbf{if } data = \_ \bullet (\_, t) \textbf{ then } t \textbf{ else } 0 \\
update(m, t, data) &\triangleq data \bullet (m(), t) \\
isQuorum(votes, t, cmd) &\triangleq |votes| \geq f + 1
\end{aligned}
$$

Vertical Paxos [Lamport et al. 2009] is a version of Multi Paxos that allows different quorum sizes in the `elect` and `commit` phases as long as they still overlap. For example, the `elect` phase could use a quorum of size $2f + 1$, which allows `commit` to only require 1 vote. We model this with a `conf` parameter to query the configuration (the quorum sizes) at a particular logical time. The other parameters are the same as for Multi Paxos.

$$
\begin{aligned}
conf &: \mathbb{N}_{time} \to (\mathbb{Z} * \mathbb{Z}) \\
rdata &\triangleq List(\mathbb{Z} * \mathbb{N}_{time}) \\
rtime(data) &\triangleq \textbf{if } data = \_ \bullet (\_, t) \textbf{ then } t \textbf{ else } 0 \\
update(m, t, data) &\triangleq data \bullet (m(), t) \\
isQuorum(votes, t, cmd) &\triangleq \textbf{let } (esize, csize) = conf\ t \textbf{ in} \\
&\qquad |votes| \geq \textbf{if } cmd = Elect \textbf{ then } esize \textbf{ else } csize
\end{aligned}
$$

CASPaxos [Rystsov 2018] updates the replicated state in-place rather than keeping a log, which can minimize packet sizes, and eliminates the need for operations like log compaction. Instead of replicating a concrete state, replicas store a "change function", which can compute the current state by applying it to an initial value. This is very similar to the update function, the primary difference being that an update function is a purely logical abstraction (i.e., it does not actually exist at runtime), whereas CASPaxos replicas do physically store and communicate change functions.

$$
\begin{aligned}
\mathit{rdata} &\triangleq ((\mathbb{Z} \to \mathbb{Z}) * \mathbb{N}_{\mathit{time}}) \mid \bot \\
\mathit{rtime}(\mathit{data}) &\triangleq \textbf{if } \mathit{data} = (\_, t) \textbf{ then } t \textbf{ else } 0 \\
\mathit{update}(m, t, \mathit{data}) &\triangleq \textbf{if } \mathit{data} = (f, t) \textbf{ then } (\lambda x.\, m(f(x)), t) \textbf{ else } (m, t) \\
\mathit{isQuorum}(\mathit{votes}, t, \mathit{cmd}) &\triangleq |\mathit{votes}| \geq f + 1
\end{aligned}
$$

4.5.4 Primary Backup

We have focused mainly on Paxos-like systems, but Advert also supports primary backup protocols such as Chain Replication [van Renesse and Schneider 2004], and CRAQ [Terrace and Freedman 2009]. These protocols operate in phases that are analogous to the three consensus phases, but rather than collecting quorums of votes, they ensure consistency by passing commands along a chain of replicas. Once a command reaches the tail of the chain, it is guaranteed to have been approved by every other replica, so it is considered committed. Instead of an election phase, clients can ensure they have the most up-to-date state by reading from the tail.

Interestingly, despite the different communication patterns, these protocols can still be described by the ADO model's `pull`, `invoke`, and `push` operations. In fact, our refinement proof for Chain Replication shares many key elements with the proof for Paxos, such as logical time reordering and network completion. By proving that both Paxos and Chain Replication refine Advert, we are showing that both satisfy a common high-level interface, which can be used to build protocol-agnostic distributed applications.

4.5.5 C Implementations

To show that ADVERT enables end-to-end verification, we implemented Multi Paxos in C and proved it correct with respect to its network specification using certified concurrent abstraction layers (CCAL) [Gu et al. 2018]. The details of this proof are largely unrelated to the ADO model and follow the approach of prior work, including CertiKOS [Gu et al. 2016] and WormSpace [Shin et al. 2019].

The general strategy is to divide the code into small modules, or *layers*, which consist of some private state and a public interface. Each method of the interface has both a C implementation and an abstract Coq specification. The C implementation is embedded in Coq using the CompCert compiler's [Leroy 2009] Clight abstract syntax tree and semantics. The Clight code is shown to be correct with respect to the Coq specification by proving memory safety (i.e., there are no null pointer dereferences or out-of-bounds array accesses) and functional correctness (i.e., the low-level, in-memory state representation matches the abstract representation).

Layers can be composed by combining the abstract interface of an existing layer with some newly introduced C private state and functions. Compared to the network-level refinement proofs, these functional correctness proofs are sometimes time consuming, but generally fairly mechanical and unsurprising. Together, these proofs provide a formal connection between ADO specifications of applications like the key-value stores to efficient executable code.

4.6 Evaluation

Verification Effort The ADVERT codebase consists of approximately 2K lines of Coq specifications and 18K of safety and refinement proofs (5K for Paxos, 2K for Chain Replication, and 11K of shared libraries). Thanks to the reusable proof template, instantiating four Paxos variants from the generic network specification takes only 340 lines. The specifications and proofs of the DApps and ADOs in Section 4.4 take 680 lines for the key-value stores and 470 lines for 2PC. The 2.6K lines of Multi Paxos C code require 43.9K lines of functional correctness proofs, which could be significantly reduced through automation [Sjöberg et al. 2019]. The code is verified using CompCert's Clight semantics [Leroy 2009] and runs on both Linux and CertiKOS [Gu et al. 2016] (augmented with unverified `send` and `recv` system calls).

The amount of developer effort required to use ADVERT depends on the level at which one wants to reason. To verify an application end-to-end there are three, mostly orthogonal steps: writing the ADO and DApp specifications, proving that the network-level protocol refines the ADO model, and proving that the C implementation refines the network-level protocol. If one reuses an existing verified network-level protocol, then the second and third steps can be skipped. One might also be satisfied with reasoning about a high-level model, in which case the ADO specifications alone are sufficient.

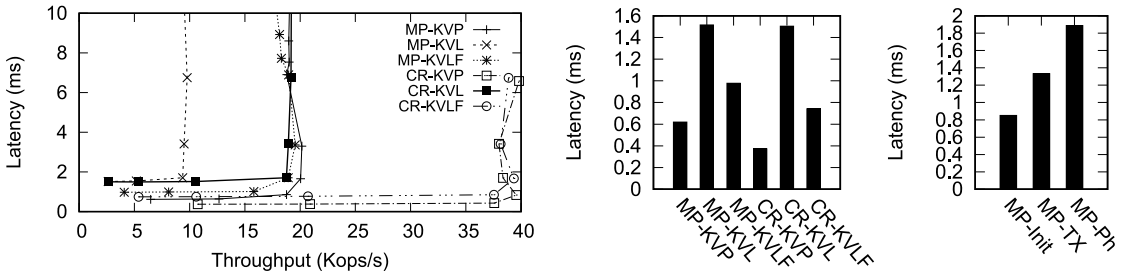
The most challenging step is the refinement between network and ADO-level specifications, though it is only required once per protocol, and similarities between protocols can be exploited to reduce the proof effort. This step can also be significantly simplified by lifting the proof to the ADO level, which we demonstrate in Chapters 5 and 6. Verify-

ing a C implementation is also quite laborious, though it is typically conceptually more straightforward than the network-ADO refinement. Advert is not tied to a specific C verification framework because all that matters is that the code can be abstracted to a network-based model in the style described in Section 4.5.1. Therefore, although we use CCAL, one could instead choose, for example, the Verified Software Toolchain [Appel 2011] or RefinedC [Sammler et al. 2021].

By comparison, working with ADO and DApp specifications is much simpler. Much of the network level's complexity is hidden and one can treat ADOs almost as standard concurrent objects, albeit with a somewhat different failure model. The degree of difficulty of composing ADOs is application-dependent; however, `KVLockFree` demonstrates that even fairly sophisticated composition patterns are feasible. In practice, many modern distributed systems rely on a coordinator to order operations across independent objects [Dean 2009] (e.g., the microservice [Killalea 2016] and serverless computing [Castro et al. 2019] paradigms). By building up a library of reusable components (e.g., locks, transactions), it would be straightforward to express many of these systems in Advert.

Performance Figure 4.19 shows latency and throughput measurements of C implementations of the key-value store (KVS) and 2PC designs from Section 4.4. KVS benchmarks use Multi Paxos and Chain Replication while 2PC only uses Multi Paxos. The experiments were run in Amazon EC2 with three replicas per ADO. We only vary the write workload, as reads can be optimized with extra learner/cache servers.

For key-value store designs (Figures 4.19a and 4.19b), `KVPrimitive` exhibits the lowest latency and the highest throughput, but cannot separate metadata and data for modularity



(a) KVS latency vs throughput. **(b)** KVS steady state latency. **(c)** 2PC latency.

Figure 4.19: Performance of different key-value store (KVS) and 2PC designs.
(MP = Multi Paxos, CR = Chain Replication, KVP = `KVPrimitive`, KVL = `KVLock`, KVLF = `KVLockFree`, Init = calling `pull` upon Init, TX = calling `pull` per TX request, Ph = calling `pull` per each 2PC phase)

and manageability. `KVLock`'s lock creates a performance bottleneck as each request accesses it twice (`tryAcquire` and `release`). The best compromise is `KVLockFree` where metadata and data are managed separately with only a moderate increase in latency but the same throughput as `KVPrimitive`.

Comparing across protocols, Chain Replication's serial communication achieves higher throughput than Multi Paxos's broadcasting approach; however, Multi Paxos can make progress with only $f + 1$ out of $2f + 1$ replicas, whereas Chain Replication must halt for reconfiguration after even one failure. Comparisons with unverified, open source Multi Paxos [Moraru et al. 2013] and Chain Replication [Balakrishnan et al. 2012; CorfuDB 2017] implementations show that our code achieves higher peak throughputs (13 Kops/s vs. 20 Kops/s for Multi Paxos and 7 Kops/s vs. 39 Kops/s for Chain Replication). Our code also outperforms IronFleet's IronRSL [Hawblitzel et al. 2015a], which was found to have a lower throughput than this same Multi Paxos implementation. Note that these systems are implemented in different languages so our point is not to claim better performance, but to demonstrate that using Advert does not inherently limit efficiency. The more

interesting takeaway is that different implementations of the same application can exhibit different performance and reliability characteristics while still sharing a common ADO-level specification.

ADOs also support performance tuning by adjusting method-calling patterns. Figure 4.19c shows transaction processing latencies of 2PC designs (Figure 4.11) with three RMs in which `pull` is called in different places: once during `init`; once on every transaction request (`handle_request`); and once for each phase of 2PC (twice per `handle_request`). Under this experiment exactly the same tasks are executed, but the performance varies up to 2X for different designs. Our aim here is to show that these design choices, which are invisible in a conventional SMR-like API, can significantly affect performance, and the ADO model allows developers to easily experiment with them.

4.7 Summary

Advert is an instance of the ADO model designed for compositional reasoning about distributed applications that hides unnecessary implementation-level complexities while faithfully capturing common high-level distributed behaviors and failure cases. It can be connected to network-level specifications of protocols such as Paxos and Chain Replication through contextual refinement, and the clean separation between implementation and specification allows one to change an application's underlying implementation without modifying ADO-level specifications or proofs. We took advantage of this implementation flexibility to build three versions of a key-value store, including a lock-free implementation. By exposing certain failure cases, the ADO model supports a wider range of method-calling

patterns and optimizations than SMR, which we used to build 2PC from a composition of replicated RMs.

Although Advert works nicely for these types of distributed applications, it is less useful for proving protocol-level properties such as safety. Chapter 5 demonstrates that this is not a fundamental limitation of the ADO model and that different variants can cover a large portion of the abstraction spectrum (Figure 1.1).

Chapter 5

Adore: Atomic Distributed Objects with Reconfiguration

This chapter discusses the Adore variant of the ADO model, which is designed to support general protocol-level safety proofs. Notably, it has first-class support for a generic reconfiguration scheme, an important but often overlooked operation. Section 5.1 summarizes the challenges of verifying reconfiguration. Section 5.2 then gives an intuitive overview of Adore's design, followed by a formal presentation in Section 5.3. Section 5.4 discusses some important steps of the safety proof, with a focus on subtle problems that Adore is uniquely suited to solve. Sections 5.5 and 5.6 then demonstrate Adore's generality by instantiating it with different protocols and reconfiguration schemes. Finally, Section 5.7 evaluates the proof effort and Section 5.8 summarizes the results.

5.1 Motivation

Server failures are inevitable in a distributed setting [Gill et al. 2011; Meza et al. 2018], so a method for safely and efficiently replacing replicas is essential. However, it is essential to maintain the invariant that elections and commits have overlapping quorums, so such configuration changes must be handled carefully. There are many algorithms to accomplish this [Lamport et al. 2008]. Some, such as Stoppable Paxos [Malkhi et al. 2008], use a "stop-the-world" approach that first blocks new commands from being committed by the old configuration, then copies the logs to the new configuration, and finally resumes normal processing. This somewhat simplifies the problem by ensuring that at no point are both configurations active at once; however, it also incurs a performance cost due to the disruption in service.

An alternative is "hot" reconfiguration, which dynamically alters the configuration without blocking the normal processing of commands. Although clearly a more attractive option, it introduces additional complexities by temporarily intermingling the old and new configurations. This obscures the fundamental invariants on which consensus protocols rely, which makes proving their safety significantly more challenging. Adore demonstrates that, with the proper abstraction, the additional complexity can be managed by hiding all but the essential details. As an added benefit, this allows for a more generic model that can support a variety of implementations.

As an example of a hot reconfiguration scheme we first explain Raft's single-server membership change algorithm [Ongaro 2014]. This example also highlights the challenges of reconfiguration and the need for formal verification because, despite expert review and

an implementation in a production environment, this algorithm, as originally presented, contained a critical bug [Ongaro 2015].

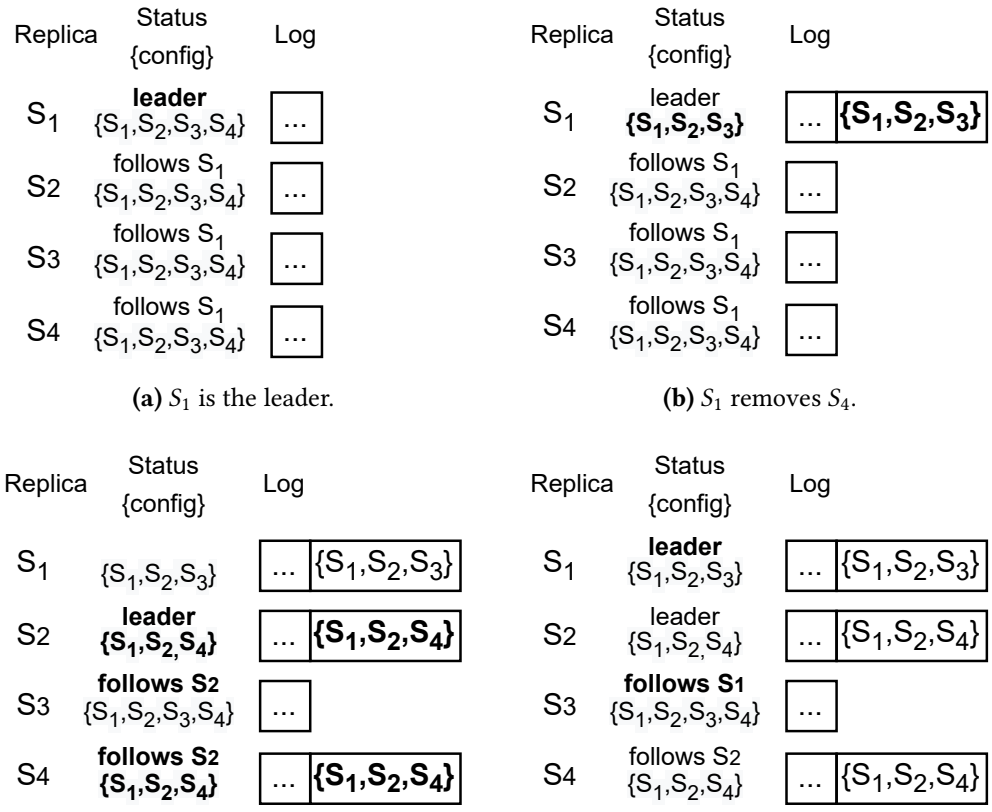
Raft's Flawed Approach The core idea of the algorithm is to communicate membership changes through the usual log replication machinery using a special command. The key difference between the special and regular commands is the latter are applied only after they are committed, while the former take effect immediately upon entering a replica's log. This speculative execution allows new replicas to begin participating sooner, but, because uncommitted commands may be overwritten, it requires special care. Therefore, two conditions must be met before a leader may propose a new configuration.

R1 A new configuration can differ from the leader's configuration by at most one replica.

R2 The leader's log cannot contain any uncommitted reconfiguration commands.

These restrictions are meant to ensure that consecutive configurations still have overlapping quorums, so the usual (static configuration) safety arguments still hold. R1 guarantees that a majority subset of a new configuration still shares at least one replica with the old one, and R2 makes sure configurations only change once before being committed. At first glance these seem sufficient; however, they miss a subtle but critical corner case that took nearly a year to discover.

The Problem Consider Figure 5.1, in which the configuration is S_1–S_4 and S_1 is the leader. S_1 proposes a new configuration that removes S_4, but fails to replicate it. S_2 then initiates an election and becomes the leader with the support of S_3 and S_4. It is unaware of S_1's reconfiguration attempt, so it may propose its own that removes S_3 instead. S_2 begins



(a) S_1 is the leader.

(b) S_1 removes S_4.

(c) S_2 is elected leader, removes S_3 and commits.

(d) S_1 is elected with the old configuration.

Figure 5.1: Raft's reconfiguration can violate safety.

using its new configuration immediately, so the command is committed once it reaches S_4 ($\{S_2, S_4\}$ is a majority of $\{S_1, S_2, S_4\}$).

Suppose now that S_1 initiates another election and receives votes from itself and S_3. Because it also uses the latest configuration from its log ($\{S_1, S_2, S_3\}$), these two votes constitute a quorum and it wins the election. At this point the game is lost because S_1 and S_2 are leaders with disjoint quorums, which means S_1 may commit a command without the approval of S_2 or S_4 and S_2 may do so without S_1 or S_3. This can cause the committed log entries to diverge, violating the safety guarantee.

The problem is that reconfiguration and consensus are inherently circularly related,

and this approach fails to account for the effect this has on elections. In particular, although R2 prevents a leader from changing the configuration until the current one is committed, it does not stop other leaders from being elected under uncommitted configurations.

The Solution The solution proposed by Ongaro [2015] is to invalidate all pending reconfiguration commands before issuing new ones. This can be accomplished by committing a command with the previous configuration. Suppose that, in Figure 5.1, after S_2 becomes the leader, before proposing a new configuration, it commits a regular method under the original configuration ($\{S_1, S_2, S_3, S_4\}$). This would require at least three of the replicas to update their logs, which means S_1's log is no longer sufficiently up-to-date, and it would be unable to win an election, thereby preventing the diverging configurations.

This solution seems to avoid the safety issue, so a third condition is added that must be met before proposing a new configuration.

R3 The leader's log must contain a committed command with the current timestamp.

Ongaro [2015] gives a very high-level proof sketch that R3 solves the problem in general by arguing that a leader cannot be elected without the latest committed command in its log. The sketch enumerates the possible configurations for the leader and command and shows that in each case their quorums must overlap because of R1–3.

As before, this argument seems reasonable, but it overlooks some subtle issues. It relies on the invariant that leaders are elected with unique timestamps, which is straightforward to prove with static configurations, but becomes more subtle with reconfiguration as the leaders' vote quorums may no longer overlap. As it turns out, the conclusion of the proof is correct, but Section 5.4 shows how the wrong approach can lead to circular reasoning

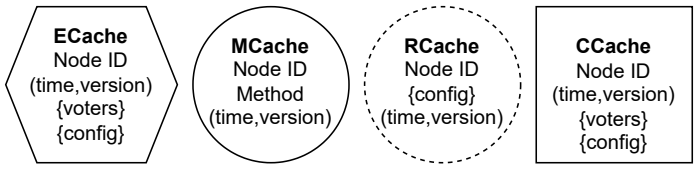
and why a model that cleanly exposes reconfiguration's mutual relation with consensus is necessary to avoid it.

5.2 Overview

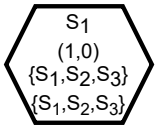
Additional Cache Metadata Adore uses the same basic ADO concepts as Advert; namely, a cache tree and the `pull`, `invoke`, and `push` operations. However, whereas Advert intentionally omits details like the current configuration that fall outside its scope, Adore must track this information in the cache tree in order to reason about safety properties. In particular, caches are now annotated with their voters, as well as the configuration under which they were created.

Additionally, a new `reconfig` operation is introduced along with a corresponding *RCache* to model reconfiguration. Following Raft's single-server approach, these behave almost identically to `invoke` and *MCaches*, except that there are limits to when `reconfig` may be called and an *RCache* contains a new configuration instead of a method. Every other type of cache inherits its configuration from its parent, which models the speculative behavior where a replica uses the latest configuration in its log even if it is uncommitted.

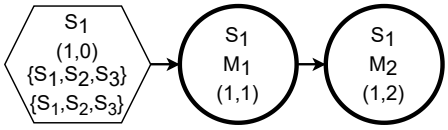
Figure 5.2 shows the evolution of a system through a sequence of operations similar to Figure 3.2, but with the added cache metadata. In Figure 5.2b, S_1 calls `pull` and successfully becomes the leader with votes from every replica (S_1, S_2, S_3). It then invokes methods M_1 and M_2, creating *MCache* nodes on its active branch (Figure 5.2c). For the moment, these methods are only in S_1's log, so, unlike *ECaches* and *CCaches*, the *MCaches* do not contain a set of voters. When it calls `push`, the oracle indicates that M_1 was received by



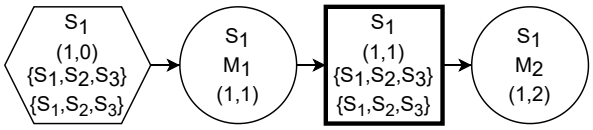
(a) Key for ADORE caches.



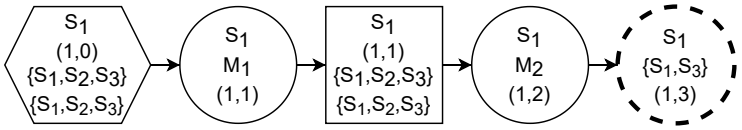
(b) S_1 calls `pull`.



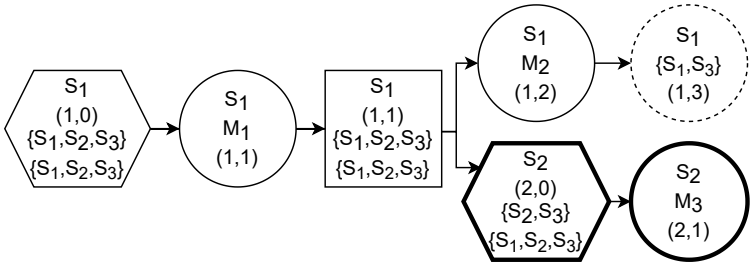
(c) S_1 invokes two methods.



(d) S_1 calls `push` and commits only M_1.



(e) S_1 removes S_2 from the configuration.



(f) S_2 calls `pull` and invokes M_3.

Figure 5.2: Sample ADORE behaviors.

every replica, but that M_2 failed to reach a quorum. A *CCache* is created between M_1 and M_2 to reflect this partially successful commit (Figure 5.2d).

Next, S_1 attempts to remove S_2 from the set of replicas by calling `reconfig` with the new configuration $\{S_1, S_3\}$. Like invoking a method, this adds an *RCache* leaf to the active branch (Figure 5.2e). Before S_1 is able to commit its new configuration, S_2 preempts it by calling `pull` and receiving votes from S_2 and S_3 (Figure 5.2f). The new *ECache* is inserted after the most up-to-date cache observed by any of the election voters, which in this case is S_1's *CCache*. Although both the *MCache* containing M_2 and the *RCache* are more recent, S_1 has not yet managed to replicate them to either S_2 or S_3. Finally, S_2 continues to extend the new branch with its own methods.

Generality The cache tree abstraction may seem far removed from a protocol's actual implementation, but it is really the same information restructured to highlight the important details. We can prove this with refinement, which implies that Adore captures every valid behavior of a network-based model of a protocol, and therefore Adore's safety properties hold for the protocol as well. Adore's `pull`, `push`, `invoke`, and `reconfig` operations map fairly directly onto the election, commit, and local log update phases found in most consensus protocols, so, in fact, this relation can be proved for many protocols, including various Paxos variants and Raft. Section 5.5 demonstrates this in more detail.

The other dimension in which Adore is general is its reconfiguration scheme. Just as Raft's single-server algorithm requires R1–3 to hold before reconfiguring, Adore has similar conditions that must be met. However, we observe that R1 is stronger than necessary; all that is needed is that consecutive configurations have overlapping quorums.

$$
\begin{aligned}
\mathit{Cache} &\triangleq \mathit{ECache}(\mathbb{N}_{nid} * \mathbb{N}_{time} * \mathbb{N}_{vrsn} * \mathit{Set}(\mathbb{N}_{nid}) * \boxed{\mathit{Config}}) \\
&\mid \mathit{MCache}(\mathbb{N}_{nid} * \mathbb{N}_{time} * \mathbb{N}_{vrsn} * \mathit{Method} * \boxed{\mathit{Config}}) \\
&\mid \boxed{\mathit{RCache}(\mathbb{N}_{nid} * \mathbb{N}_{time} * \mathbb{N}_{vrsn} * \mathit{Config})} \\
&\mid \mathit{CCache}(\mathbb{N}_{nid} * \mathbb{N}_{time} * \mathbb{N}_{vrsn} * \mathit{Set}(\mathbb{N}_{nid}) * \boxed{\mathit{Config}}) \\
\mathit{CacheTree} &\triangleq \mathbb{N}_{cid} \rightharpoonup \mathbb{N}_{cid} * \mathit{Cache} \\
\mathit{TimeMap} &\triangleq \mathbb{N}_{nid} \rightharpoonup \mathbb{N}_{time} \\
\Sigma &\triangleq \mathit{CacheTree} * \mathit{TimeMap}
\end{aligned}
$$

Figure 5.3: ADORE state definitions.

In fact, the protocol's safety is completely independent from the definitions of quorum and valid configuration as long as they guarantee this property. By parameterizing these features ADORE becomes a generic verification framework that permits many possible implementations (see Section 5.6 for examples).

5.3 ADORE Formal Semantics

This section formalizes the previous intuitive description of ADORE. We mark everything related to reconfiguration in blue with a box. Removing these parts leaves a configuration-aware model (CADO) that is also useful for reasoning about the safety of protocols with static configurations. ADORE shares many features with ADVERT, but, in the interest of completeness, they are explained here again.

State Figure 5.3 defines the type Σ for state (st), which is a pair of a cache tree ($tree$), and the largest timestamp that each replica has observed ($times$). As before, we use the notation $name(st)$ to represent extracting one of these fields (e.g., $tree(st)$ returns the first element). Figure 5.4 declares that the type of the configuration ($Config$) is an opaque parameter with functions to extract a set of replicas ($mbrs$) and decide if some set constitutes a quorum

Parameters

$$Config : Type \qquad isQuorum : Set(\mathbb{N}_{nid}) \to Config \to \mathbb{B}$$
$$mbrs : Config \to Set(\mathbb{N}_{nid}) \qquad \mathsf{R1}^{+} : Config \to Config \to \mathbb{B}$$

Assumptions about $\mathsf{R1}^{+}$ and *isQuorum*

$$\text{(Reflexive)}\ \mathsf{R1}^{+}(cf, cf)$$
$$\text{(Overlap)}\ \mathsf{R1}^{+}(cf, cf') \land isQuorum(Q, cf) \land isQuorum(Q', cf') \implies Q \cap Q' \neq \emptyset$$

Definitions

$$R2(tr, C) \triangleq \forall C' \in tr.\, C' = RCache(\_) \land C' \uparrow C \implies \exists C'' \in tr.\, C'' = CCache(\_) \land C' \uparrow C'' \land C'' \uparrow C$$
$$R3(tr, C) \triangleq \exists C' \in tr.\, C' = CCache(\_) \land time(C') = time(C) \land C' \uparrow C$$
$$canReconf(tr, C, ncf) \triangleq \mathsf{R1}^{+}(conf(C), ncf) \land R2(tr, C) \land R3(tr, C)$$

Figure 5.4: Configuration/quorum parameters and definitions.

(*isQuorum*). This allows the model and safety proof to work for any instantiation of these parameters as long as they satisfy the Reflexive and Overlap invariants.

Caches Caches are divided into election (*ECache*), method (*MCache*), reconfiguration (*RCache*), and commit (*CCache*) variants. Each has a unique cache ID (*cid*) and the cache tree is implemented as a partial map from *cid* to the corresponding cache, plus the *cid* of the cache's parent (with 0 reserved for the root). The functions for growing the tree are *addLeaf*, which adds a child to a parent, and *insertBtw*, which inserts a cache between a parent and its children.

Each type of cache contains the node ID (*nid*) of the replica that called the operation creating it (*caller*), a timestamp (*time*), a version number (*vrsn*), and the configuration (*conf*)

$$
\begin{aligned}
Op \triangleq\ & \texttt{pull} : \mathbb{N}_{nid} \to \Sigma \to \Sigma \\
\mid\ & \texttt{invoke} : \mathbb{N}_{nid} \to \mathit{Method} \to \Sigma \to \Sigma \\
\mid\ & \boxed{\texttt{reconfig} : \mathbb{N}_{nid} \to \mathit{Config} \to \Sigma \to \Sigma} \\
\mid\ & \texttt{push} : \mathbb{N}_{nid} \to \Sigma \to \Sigma
\end{aligned}
$$

Figure 5.5: ADORE operations.

under which it was called (the root cache is initialized with some $conf_0$). The timestamp corresponds to a Paxos ballot number or a Raft term number and is assigned to a leader in each round. The version number resets to 0 at the start of each round and increments on every method/reconfiguration call. In Figure 5.6, we define a strict order ($\succ$) on caches by comparing the lexicographic order of their timestamp and version number pairs, with the exception that if a *CCache* has the same timestamp and version as a non-*CCache*, the *CCache* is considered greater, which is needed to make $\succ$ a total order.

ECaches and *CCaches* are also annotated with the node IDs of their *voters*; i.e., the set of replicas that voted for them. An *MCache* or *RCache*'s only voter is its caller. Every cache is also associated with a related, but subtly different set of replicas called its *supporters*. For *MCaches*, *RCaches*, and *CCaches* these sets are the same, but an *ECache*'s only supporter is its caller. The difference is related to a replica's *active* cache, which is its current view of the latest state (this corresponds to its local log). A replica may *vote* for a cache during an election, but it does not yet make it *active*. Only when a replica votes to commit a cache does it have enough evidence to know it is safe to also *support* the cache and make it active.

Finally, *MCaches* contain a *Method*. In practice, this encodes an application-specific function (e.g., increment a counter) to be applied once committed; however, as the method bodies have no bearing on the protocol's safety, we simply treat them as opaque identifiers.

$$
\begin{aligned}
C \uparrow C' &\triangleq C = parent(C') \vee C \uparrow parent(C') \\
C_1 \succ C_2 &\triangleq (time(C_1), vrsn(C_1)) > (time(C_2), vrsn(C_2)) \\
&\quad \vee ((time(C_1), vrsn(C_1)) = (time(C_2), vrsn(C_2)) \\
&\quad\quad \wedge C_1 = CCache(\_) \wedge C_2 \neq CCache(\_)) \\
voters(C) &\triangleq \textbf{if } C = ECache(\_,\_,\_,Q,\_) \textbf{ then } Q \textbf{ else} \\
&\quad \textbf{if } C = MCache(nid,\_,\_,\_,\_) \textbf{ then } \{nid\} \textbf{ else} \\
&\quad \textbf{if } C = RCache(nid,\_,\_,\_) \textbf{ then } \{nid\} \textbf{ else} \\
&\quad \textbf{if } C = CCache(\_,\_,\_,Q,\_) \textbf{ then } Q \\
supporters(C) &\triangleq \textbf{if } C = ECache(nid,\_,\_,\_,\_) \textbf{ then } \{nid\} \textbf{ else } voters(C) \\
freshCID(tr) &\triangleq \max\{cid(C) \mid C \in tr\} + 1 \\
addLeaf(st, C_P, C_{new}) &\triangleq (tree(st)[freshCID(tree(st)) \mapsto (C_P, C_{new})], times(st)) \\
insertBtw(st, C_P, C_{new}) &\triangleq \textbf{let } tr = tree(st) \textbf{ in} \\
&\quad \textbf{let } tr' = tr[cid(C) \mapsto (cid(C_{new}), C_{new}) \mid \forall(\_, C) \in tr] \textbf{ in} \\
&\quad (tr'[freshCID(tree(st)) \mapsto (C_P, C_{new})], times(st)) \\
setTimes(st, Q, t) &\triangleq (tree(st), times(st)[s \mapsto t \mid \forall s \in Q]) \\
isLeader(st, nid, C) &\triangleq times(st)[nid] = time(C) \wedge caller(C) = nid \\
validVotes(nid, Q, C) &\triangleq nid \in Q \wedge Q \subseteq mbrs(conf(C)) \\
active(tr, nid) &\triangleq \max_{\succ}\{C \in tr \mid nid \in supporters(C)\} \\
activeC(tr, nid) &\triangleq \max_{\succ}\{C \in tr \mid nid \in supporters(C) \wedge C = CCache(\_)\} \\
maxActive(tr, Q) &\triangleq \max_{\succ}\{C \in tr \mid Q \cap supporters(C) \neq \emptyset\} \\
canCommit(C, nid, st) &\triangleq \big(C = MCache(\_) \vee \boxed{C = RCache(\_)}\big) \\
&\quad \wedge isLeader(st, nid, C) \wedge C \succ activeC(tree(st), nid)
\end{aligned}
$$

Figure 5.6: ADORE auxiliary definitions.

Operations Each of ADORE's operations (`pull`, `invoke`, `reconfig`, and `push`) takes its caller's node ID, the current state (Σ), and for `invoke` and `reconfig` a new method or configuration, and returns a new state (see Figure 5.5). The `pull` and `push` operations rely on an oracle $\mathbb{O}$ (consisting of $\mathbb{O}_{pull}$ and $\mathbb{O}_{push}$ respectively) to model nondeterministic network behaviors. The semantics of each operation are defined in Figure 5.7 (with helper functions defined in Figure 5.6). We write $\mathbb{O} \vdash op : st \rightsquigarrow st'$ to mean calling the operation op on state st with oracle $\mathbb{O}$ results in st'.

PullOk

$$\frac{\mathbb{O}_{pull}(st, nid) = Ok(Q, Q_{ok}, C_{max}, t) \quad st' \triangleq setTimes(st, Q, t) \quad C_{new} \triangleq ECache(nid, t, 0, Q, conf(C_{max}))}{\mathbb{O} \vdash \texttt{pull}(nid) : st \rightsquigarrow \textbf{if } Q_{ok} \textbf{ then } addLeaf(st', C_{max}, C_{new}) \textbf{ else } st'}$$

InvokeOk

$$\frac{C_A \triangleq active(tree(st), nid) \quad isLeader(st, nid, C_A) \quad C_{new} \triangleq MCache(nid, time(C_A), vrsn(C_A) + 1, M, conf(C_A))}{\mathbb{O} \vdash \texttt{invoke}(nid, M) : st \rightsquigarrow addLeaf(st, C_A, C_{new})}$$

ReconfigOk

$$\frac{C_A \triangleq active(tree(st), nid) \quad isLeader(st, nid, C_A) \quad canReconf(tree(st), C_A, ncf) \quad C_{new} \triangleq RCache(nid, time(C_A), vrsn(C_A) + 1, ncf)}{\mathbb{O} \vdash \texttt{reconfig}(nid, ncf) : st \rightsquigarrow addLeaf(st, C_A, C_{new})}$$

PushOk

$$\frac{\mathbb{O}_{push}(st, nid) = Ok(Q, Q_{ok}, C_M) \quad st' \triangleq setTimes(st, Q, time(C_M)) \quad C_{new} \triangleq CCache(nid, time(C_M), vrsn(C_M), Q, conf(C_M))}{\mathbb{O} \vdash \texttt{push}(nid) : st \rightsquigarrow \textbf{if } Q_{ok} \textbf{ then } insertBtw(st', C_M, C_{new}) \textbf{ else } st'}$$

NoOp

$$\frac{}{\mathbb{O} \vdash op(nid) : st \rightsquigarrow st}$$

Figure 5.7: Semantics of Adore operations. Every operation may fail and have no effect on the state (either because an oracle returns *Fail* or the preconditions are not met), so NoOp is parameterized by *op*, which can be any of `pull`, `invoke`, `reconfig`, or `push`. For `invoke` and `reconfig`, *op* is understood to also take *M* or *ncf* as an argument.

$$\mathbb{O}_{pull} : \Sigma \to \mathbb{N}_{nid} \to (Ok(Set(\mathbb{N}_{nid}) * \mathbb{B} * Cache * \mathbb{N}_{time}) \mid Fail)$$
$$\mathbb{O}_{push} : \Sigma \to \mathbb{N}_{nid} \to (Ok(Set(\mathbb{N}_{nid}) * \mathbb{B} * Cache) \mid Fail)$$

ValidPullOracle

$$\frac{validVotes(nid, Q, C_{max}) \quad Q_{ok} \triangleq isQuorum(Q, conf(C_{max})) \quad \forall s \in Q.\, times(st)[s] < t \quad C_{max} \triangleq maxActive(tree(st), Q)}{\mathbb{O}_{pull}(st, nid) = Ok(Q, Q_{ok}, C_{max}, t)}$$

ValidPushOracle

$$\frac{validVotes(nid, Q, C_M) \quad Q_{ok} \triangleq isQuorum(Q, conf(C_M)) \quad \forall s \in Q.\, times(st)[s] \leq time(C_M) \quad canCommit(C_M, nid, st)}{\mathbb{O}_{push}(st, nid) = Ok(Q, Q_{ok}, C_M)}$$

Figure 5.8: Valid `pull` and `push` oracle conditions.

Pull Recall that the purpose of the election phase is to choose a unique logical timestamp and a sufficiently up-to-date state snapshot with all of the committed commands. ADORE models this with `pull`, which relies on $\mathbb{O}_{pull}$ to simulate the network and arbitrarily decide what set of voters (Q) receive the election request. The oracle is allowed to make any decision that satisfies the conditions in Figure 5.8, which abstracts over the network details and treats it as a nondeterministic black box.

On success, the oracle chooses a set of voters and a time (t) that is strictly larger than any they have previously observed. The cache it returns (C_{max}) is the result of *maxActive*, and is the most up-to-date cache supported by any replica in Q. This guarantees that the leader learns about every committed method. The voters' timestamps are updated with *setTimes* to reflect their vote.

There are two outcomes for the cache tree depending on Q. If it is not a quorum, then the election fails and the only effect is the change in the timestamps. Otherwise, a new *ECache* child is added to C_{max}. The oracle may also return failure, in which case the state is unchanged (see NOOP in Figure 5.7). This is also a possible outcome for the other operations so it is written in terms of a general operation, *op*, which can be any of `pull`, `invoke`, `reconfig`, or `push`.

Invoke When a method M is invoked it finds the caller's active cache (C_A), which is the largest cache supported by *nid*. If the active cache's time is not equal to the caller's local time then it has been preempted by another leader and the method fails. Otherwise, a new *MCache* with an incremented version number is inserted into the tree as a child of the active cache (thus making it the new active cache).

Reconfig So far, every new cache inherits its parent's configuration. The only exception is an *RCache*, which is essentially a special kind of *MCache* that contains a new configuration (*ncf*) instead of a method. *RCaches* are created by `reconfig`, whose semantics are nearly identical to regular method invocation, except for a few additional restrictions, which are modified versions of R1–3 from Section 5.1. Put in words, R2 and R3 guarantee the following.

R2 There are no uncommitted *RCaches* in the active branch.

R3 There must be a *CCache* with the same timestamp as the active cache in the active branch.

As mentioned earlier, Raft's R1 is stronger than necessary, so it is replaced by the more general R1$^+$ predicate, which can be instantiated by any condition that satisfies the Reflexive and Overlap properties in Figure 5.4. Section 5.6 demonstrates a few of the many possible reconfiguration schemes this permits.

Push As with Advert, a successful `push` commits an arbitrary prefix of the most recent uncommitted commands. This is chosen by $\mathbb{O}_{push}$, which returns a cache (C_M) that satisfies *canCommit*. This means C_M must be an *MCache* or *RCache* that was called by *nid* with its current timestamp, and is more recent than the latest *CCache* supported by *nid*. This guarantees that *nid* is a valid leader, C_M is an uncommitted command, and committing it will not conflict with a previous commit. Like `pull`, the voters' timestamps must not be greater than C_M's, though they may be equal.

As with `pull`, `push` updates the voters' timestamps, and the cache tree if Q is a quorum.

The new *CCache* (C_{new}) is added with *insertBtw* rather than *addLeaf*, which puts C_{new} between C_M and its children instead of creating a leaf node. The children represent partial failures that may still be committed later on, so shifting them after the *CCache* leaves them as viable candidates for some later `pull` or `push`.

5.4 Safety Proof

The primary purpose of Adore is to simplify the verification of safety properties of consensus protocols even with the complexity introduced by reconfiguration. This section demonstrates how it accomplishes this goal by sketching the proof of replicated state safety and highlighting interesting challenges. For clarity, this section sticks mainly to informal arguments, but more rigorous proofs can be found in Appendix C.2 and in the Coq source code [Honoré et al. 2022b].

5.4.1 Breaking Circularity with rdist

Replicated state safety guarantees that clients observe the committed commands in the same order regardless of which replica they contact. This means if two replicas commit a command in a certain slot, the prefixes of their logs up to that slot are equal. Phrased in Adore terms, there exists a single branch that contains every *CCache*.

Definition 4 (Replicated State Safety)**.** *There exists a linear path from the root of the tree to a leaf node that contains every committed method. In other words, for any CCaches C_{C1} and C_{C2}, one must be a descendant of the other.*

Without reconfiguration, the core of the proof of this property is that consecutive

elections and commits both require a quorum of supporters. This implies that they share at least one replica, which ensures that a leader's log must be sufficiently up-to-date and contains the latest committed method. Note, however, that this latest commit is unique only if every committed method has a distinct timestamp and version number pair, which is easy to prove as long as every leader has a unique timestamp. The standard proof of this property reasons that leaders require a quorum of voters, so two elections must involve at least one common voter. Then, because replicas only vote for candidates with timestamps greater than what they have seen, the shared voter cannot have voted for two candidates with the same timestamp.

Now consider what happens with reconfiguration. We can no longer assume that two leaders were elected under the same configuration, so the existence of a shared voter is not automatically guaranteed. Instead, we must prove that the leaders' configurations cannot diverge to the point that they no longer overlap. $\mathsf{R1}^+$ guarantees this if the leaders are separated by only one reconfiguration, but it does not help for two or more changes. For those cases, we need R2 and R3 to show that if both leaders have the latest committed method then their configurations must still be similar. However, recall that, for there to be a unique latest committed method, leaders must have unique timestamps. This, in turn, requires their quorums to overlap, which is where we began.

This circularity arises because there may be arbitrarily many reconfigurations between the commit and election. The solution is to count the number of reconfigurations between two commands, which we call their *rdist*, and reduce the problem to smaller, more manageable steps. This is a fairly awkward property to express in a network-based specification because one must essentially construct a tree from two logs by merging their common

prefix into a branch that forks where their tails diverge. It is much more natural in Adore because the cache tree already captures this structure.

Definition 5 (rdist)**.** *Suppose C_1 and C_2 are caches with a nearest common ancestor C_A. The rdist of C_1 and C_2 is the number of RCaches on the path between C_1 and C_2, passing through C_A, not including the endpoints.*

This is a useful metric because it counts only the reconfigurations that influence a given pair of caches. We can extend this idea by defining the *rdist* of a tree to be the maximum *rdist* between any two caches in the tree. The high-level strategy then is to use induction on *rdist* to break the safety proof down into the following cases.

rdist = 0**.** The configurations are the same, so standard arguments about overlapping quorums apply.

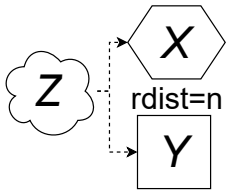
rdist = 1**.** R1$^+$, R2, and R3 guarantee that quorums still overlap.

rdist = $n + 1$**.** The cache tree must decompose into a subtree with *rdist* = n and a branch with exactly one *RCache*. The inductive hypothesis and *rdist* = 1 case guarantee *CCaches* in these regions are on the same branch.

The typical approach to proving safety in a network-based model goes by induction over the trace of network events; i.e., assume the property holds, then show it continues to hold when some replica receives a commit request, or when a candidate receives a quorum of election votes, and so on.

An advantage of Adore is that one can instead reason directly about the structure of the cache tree, which makes for simpler and more intuitive proofs. Given a cache tree, one

can consider a small number of cases that could have led to that situation, and prove that the property holds for each.

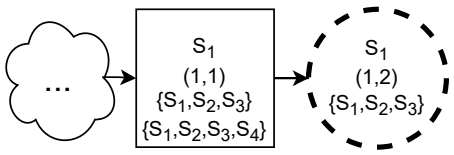


These cases are often most easily explained using pictures like the one to the left. This represents a subtree in which Z is a common ancestor of X and Y. The cloud symbol means Z can be any type of cache. The dotted arrows indicate that X and Y are descendants, but not necessarily direct children, of Z. The label $rdist = n$ means that $rdist(X, Y) = n$.

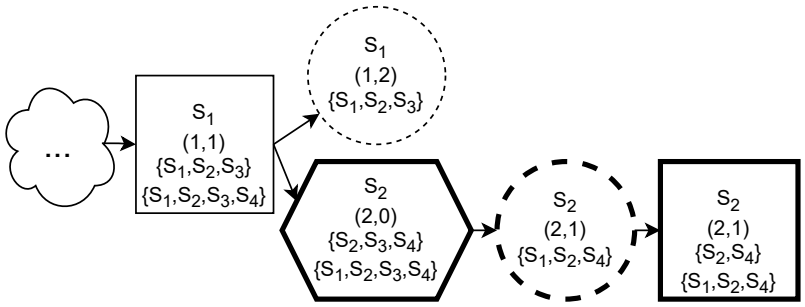
5.4.2 Base Cases

The safety proof for the $rdist = 0$ case follows the standard static-configuration argument [Lamport 2001; Ongaro and Ousterhout 2014], so we leave the details to Appendix C.2. Many properties that hold for caches where $rdist = 0$ also hold when $rdist = 1$ if one can show that their configurations still have overlapping quorums, which is precisely what R1$^+$, R2, and R3 are meant to guarantee. The purpose of R1$^+$ is clear (Overlap in Figure 5.4 ensures that when a leader proposes a new configuration it overlaps with the old one), but the other two are no less important.

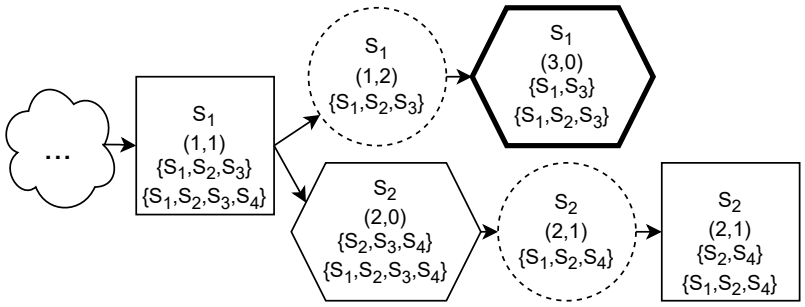
R2 ensures that a leader cannot attempt a reconfiguration while there is an uncommitted *RCache* in its branch. This prevents the configuration from changing twice in a single commit, which might break the overlap guarantee (Overlap only holds for consecutive configurations). R3 requires the leader's log to contain a committed entry with the current timestamp. This serves a similar purpose to R2 in that it prevents a leader from beginning a new reconfiguration while another leader still has one in progress. This is a particularly



(a) S_1 proposes a new configuration.



(b) S_2 commits a different configuration.



(c) S_1 is elected without S_2's *CCache*.

Figure 5.9: An example of a breach of safety without R3.

subtle problem because the new leader might not even be aware of the old reconfiguration.

For example, consider the situation in Figure 5.9 (this is the same as Figure 5.1 but with cache trees). The leader S_1 removes S_4 from the configuration but fails to commit it. Then S_2 becomes the leader, but is unaware of the reconfiguration attempt because its supporters do not include S_1, so it begins its own reconfiguration by removing S_3. It succeeds with a quorum (S_2 and S_4) of supporters. At this point, any future election must have this *CCache* in its history or else safety is compromised. However, because S_1 and S_3 did not participate in S_2's reconfiguration, S_1 is elected using its own configuration on a different branch from the *CCache*.

Note that, although S_1 and S_2 each change only one server, their configurations differ by two servers, which allows disjoint majorities. R3 prevents this because, before S_2 can remove S_3, it must commit a command with the old configuration. This blocks S_1 from being elected using its own *RCache* because S_2's *CCache* has a larger timestamp.

Together, these properties guarantee that caches with an *rdist* of 1 have overlapping quorums and therefore properties like the uniqueness of a leader's timestamp and safety follow from similar arguments to their $rdist = 0$ counterparts.

Theorem 1 (Safety, rdist ≤ 1)**.** *Any cache tree with* $rdist \leq 1$ *satisfies replicated state safety.*

5.4.3 General Case

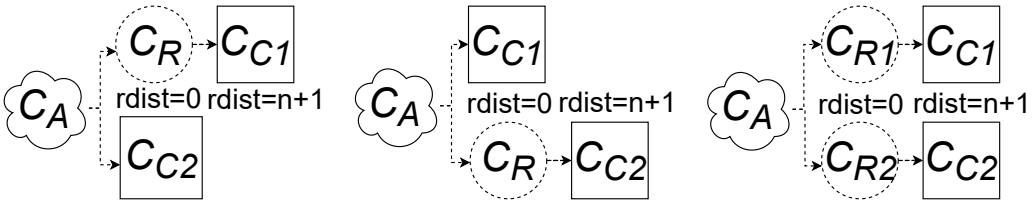
Now that we have established safety for $rdist \leq 1$, the final step is to show that the general case of $rdist = n$ reduces to a combination of these cases. To help with this reduction, we require an invariant that, as a consequence of R3, given two *RCaches* on different branches, at least one must have a *CCache* ancestor that is not an ancestor of the other.

Lemma 3 (CCache in RCache Fork)**.** *Let* C_{R1} *and* C_{R2} *be RCaches such that* $rdist(C_{R1}, C_{R2}) = 0$ *and neither is a descendant of the other, but both have a common ancestor* C_A*. Then there exists a CCache* C_C *that is a descendant of* C_A *and an ancestor of either* C_{R1} *or* C_{R2}*.*

Theorem 2 (Safety)**.** *Any cache tree,* tr*, with any rdist satisfies replicated state safety.*

Proof Sketch. We proceed by induction on $rdist(tr)$. For $rdist \leq 1$ we are done by Theorem 1. Suppose now that all trees with $rdist = n$ are safe, and $1 < rdist(tr) = n + 1$ so $1 < rdist(C_{C1}, C_{C2}) \leq n+1$ for some *CCaches* C_{C1} and C_{C2}. If $rdist(C_{C1}, C_{C2}) \leq n$, then they are in some subtree tr' with $rdist(tr') = n$, so we are done by the inductive hypothesis.

Safety also holds if C_{C1} and C_{C2} are on the same branch, and, if not, we will show that all other shapes for tr are impossible.



The first two cases are symmetric, and Lemma 3 implies that in the last case there must be another *CCache* between C_A and either C_{R1} or C_{R2}, which results in the same situation as the other cases. Therefore, we can assume without loss of generality that C_R is on the first *RCache* on C_{C1}'s branch. Let C_{CR} be C_R's first *CCache* descendant. It is enough to show that $rdist(C_{CR}, C_{C2}) \leq n$ because then C_{CR} and C_{C2} must be on the same branch, which is a contradiction. We know $rdist(C_{C1}, C_{C2}) > 1$, so C_R cannot be the only *RCache* on C_{C1}'s branch. We also know by R2 that this other *RCache* cannot be between C_R and C_{CR}. Therefore, this *RCache* does not count towards $rdist(C_{CR}, C_{C2})$ and it is at most n. □

5.5 Refinement

We now know that Adore is safe, but what does this imply for concrete protocols like Paxos and Raft? With refinement we can formalize the intuitive correspondence between Adore and these protocols and show that Adore's safety guarantees their safety. We demonstrate this for a slightly simplified version of Raft, but note that this is just one of many possible implementations.

In a sense, Adore models an abstract, synchronous version of Raft in which commu-

$$
\begin{aligned}
\Sigma_{\text{net}} &\triangleq (\mathbb{N}_{nid} \rightharpoonup \mathit{Replica}) * \mathit{Network} \\
\mathit{Replica} &\triangleq \mathbb{N}_{time} * \mathbb{N}_{vrsn} * \mathit{List}(\mathbb{N}_{time} * \mathit{Method} * \mathit{Config}) * \ldots \\
\mathit{Network} &\triangleq \mathit{Set}(\mathit{Msg}) * \mathit{Set}(\mathit{Msg}) \\
\mathit{Op}_{\text{net}} &\triangleq \texttt{elect} : \mathbb{N}_{nid} \to \Sigma_{\text{net}} \to \Sigma_{\text{net}} \\
&\quad \mid \texttt{commit} : \mathbb{N}_{nid} \to \Sigma_{\text{net}} \to \Sigma_{\text{net}} \\
&\quad \mid \texttt{invoke} : \mathbb{N}_{nid} \to \mathit{Method} \to \Sigma_{\text{net}} \to \Sigma_{\text{net}} \\
&\quad \mid \texttt{reconfig} : \mathbb{N}_{nid} \to \mathit{Config} \to \Sigma_{\text{net}} \to \Sigma_{\text{net}} \\
&\quad \mid \texttt{deliver} : \mathit{Msg} \to \Sigma_{\text{net}} \to \Sigma_{\text{net}}
\end{aligned}
$$

Figure 5.10: Selected Raft network-based state and operations.

nication happens atomically and in logical time order. Through a series of refinements we show that a more realistic asynchronous network-based specification of Raft behaves equivalently to this synchronous version. This part is similar to previous work [Chajed et al. 2018; Hawblitzel et al. 2015a; v. Gleissenthall et al. 2019], but an important distinction is the final refinement, which lifts the simplified network-based model to Adore.

Specification We begin by writing a standard asynchronous network-level specification for our Raft protocol. The state (Figure 5.10) comprises a set of replicas each with a current timestamp, a local log of commands, and some additional bookkeeping details (e.g., the current leader, number of votes received). Communication between replicas is only possible through the network, which is represented as a pair of bags of sent and delivered messages, respectively. Messages are of four types: election/commit requests/acknowledgements. Requests are generated by the `elect` and `commit` operations. Any time after being sent, a message may arrive with `deliver`, which triggers a handler based on the type of the message. A leader may also call `invoke` and `reconfig`, which are local operations that only affect its own log.

We also create another specification that is the same except for a few simplifying assumptions: only valid messages are delivered (i.e., messages that will not be ignored by their recipients for having the wrong timestamp, coming from outside the current configuration, etc.), messages are delivered in order of their logical timestamps, and requests are received and acknowledged atomically by all of the recipients at once. Unless otherwise qualified, we refer to the asynchronous model as Raft and to this simplified version as SRaft.

Refinement Relation The next step is to define a refinement relation, $\mathbb{R}$, between Raft's and Adore's state. This includes correspondences between auxiliary state, such as timestamps and leadership flags, but for Adore's safety to imply something useful about Raft, $\mathbb{R}$ must ensure that, for any replica's local log, the methods are the same as those along that replica's active branch of the cache tree.

Note that, unlike Advert, we do not have to directly prove the safety of the network-based specification, but can instead use Adore's safety result. In particular, because Adore guarantees that every replica's active branch has a common prefix of committed methods, $\mathbb{R}$ implies that the same is true in Raft of the local logs. SRaft and Raft share the same state definitions, so $\mathbb{R}$ also holds for SRaft and Adore.

Simulation Finally, we show that Adore simulates Raft by proving that, given two states related by $\mathbb{R}$, for any step that Raft can take, there is a corresponding step for Adore that preserves the relation. If the initial states are also related, then this implies that Adore captures all valid Raft behaviors and therefore its safety implies Raft's safety.

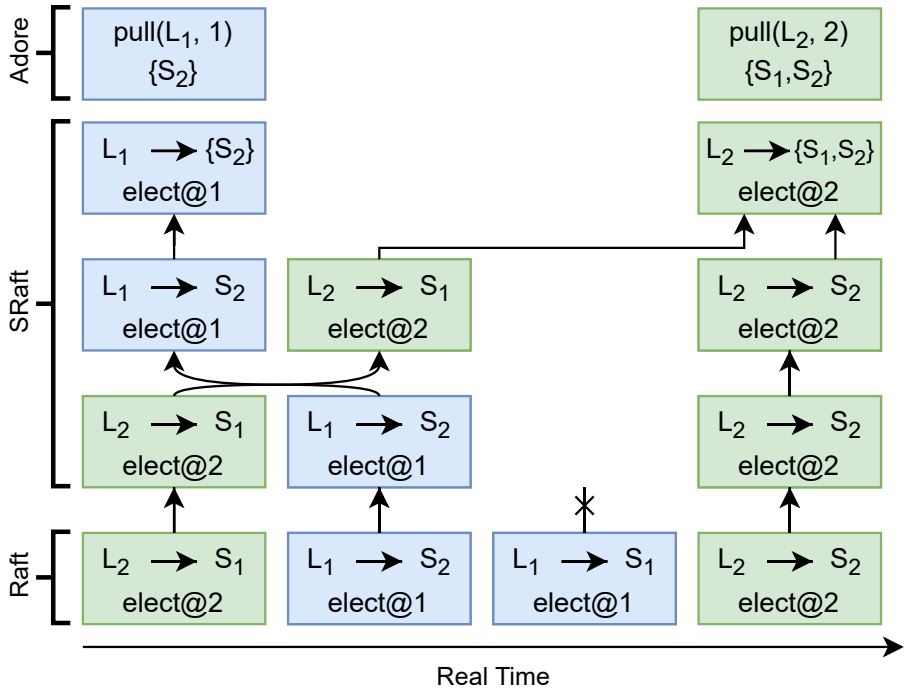


Figure 5.11: Raft to SRaft to Adore refinement. $L_1 \rightarrow S_1$ denotes a message sent from a leader to a server with the type and logical timestamp of the message indicated below.

Intuitively, the correspondence between Adore and Raft steps is clear: `pull` for elections, `push` for commits, and Adore method invocation and reconfiguration for their Raft counterparts. The reality is less straightforward because the Adore operations are atomic, while their Raft equivalents are not. SRaft's purpose is to be an intermediate specification that rearranges asynchronous Raft operations into an equivalent order that satisfies the intuitive mapping. As an example, consider the situation in Figure 5.11. We'd like to show that the four Raft receive events in the bottom layer correspond to the two `pull` requests in the top layer, but how can we be sure that these sequences of events are equivalent?

We first observe that S_1 receives a message with timestamp 1 after it received one with timestamp 2. Therefore, S_1 ignores the second message and we can safely drop it from the sequence of events. Next, we note that S_1 receiving a message has no effect on S_2 and vice-versa, so the first two receive events safely commute, which puts the sequence in

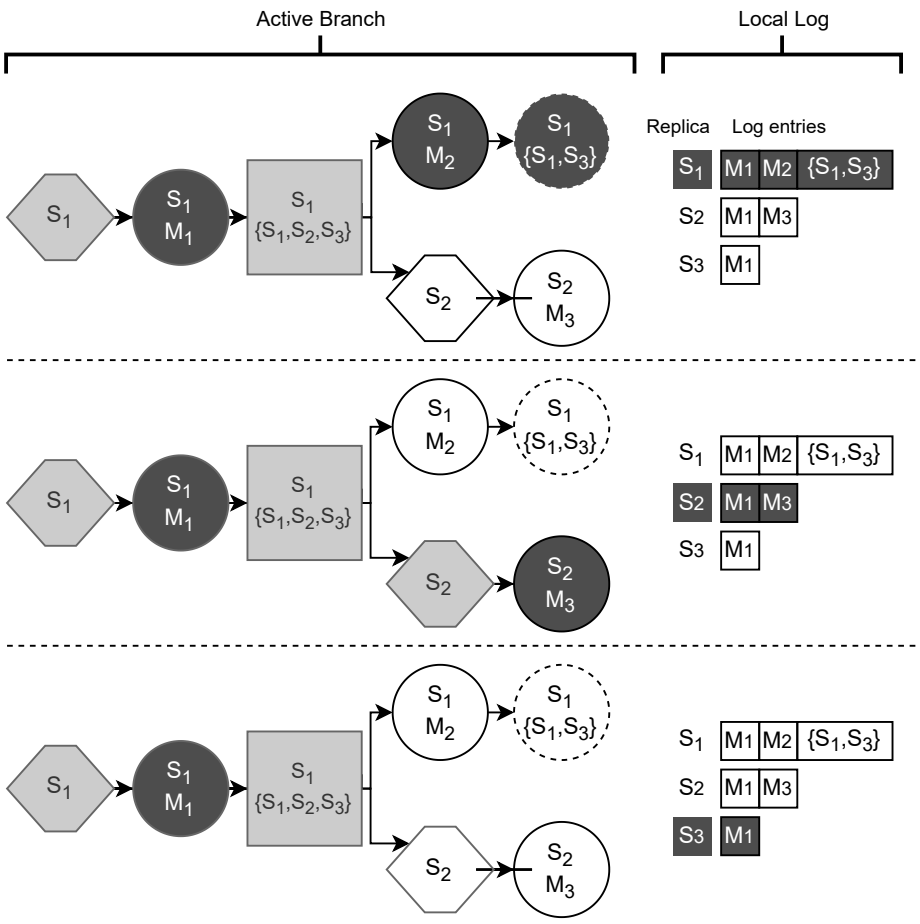


Figure 5.12: Correspondence between replicas' local logs and active branches. The active branch of each replica is shown by the gray backgrounds, with the dark gray indicating caches that have matching entries in the local log (*MCaches* and *RCaches*).

logical time order. Now that L_1 and L_2's requests are untangled, we can merge adjacent receive events and treat the messages as if they arrived at each recipient at the same time.

Now we have a much simpler network-based model with in-order, atomic message delivery that is equivalent to the asynchronous version. To complete the refinement, the final step is to show that corresponding SRaft and Adore operations preserve $\mathbb{R}$. Because we have already "lined up" the events, the bulk of the remaining work is to translate between different state representations, such as a replica's local log in SRaft and its active branch in Adore (Figure 5.12). See Appendix B.2 for more information about

these refinement layers, and the source code [Honoré et al. 2022b] for the full proofs.

5.6 Instantiating Reconfiguration Schemes

Recall that the only information about configurations that the safety proof relies on is that quorums of two configurations related by $\mathsf{R1}^+$ have a non-empty intersection. This means that, for any valid instantiation of the configuration-related parameters, the safety proof holds for free. To give a sense of how flexible ADORE's reconfiguration scheme is, we demonstrate several practical and diverse implementations.

Raft Single-Server One option is Raft's single-server algorithm, which uses a standard majority quorum and only allows configurations to add or remove one replica at a time.

$$\begin{aligned} \mathit{Config} &\triangleq \mathit{Set}(\mathbb{N}_{nid}) \\ \mathsf{R1}^+(C, C') &\triangleq C = C' \vee \exists s.\, C = C' \cup \{s\} \vee C' = C \cup \{s\} \\ \mathit{isQuorum}(S, C) &\triangleq |C| < 2 * |S \cap C| \end{aligned}$$

To see why this maintains the quorum overlap property, consider sets C and $C' = C \cup \{s\}$. A majority of C has at least $\lceil (|C| + 1)/2 \rceil = \lceil |C'|/2 \rceil$ elements, and a majority of C' has at least $\lceil (|C'| + 1)/2 \rceil$ elements, so together they total at least $(2|C'| + 1)/2$ elements. C is a subset of C', so a majority of either C or C' is a subset of C'. Given two subsets of the same set, if the sum of their cardinalities is greater than that of the superset, they must have at least one element in common. This is the case for majorities of C and C' because $(2|C'| + 1)/2 > |C'|$.

Raft Joint Consensus A more complicated case is Raft's original reconfiguration strategy [Ongaro and Ousterhout 2014], which allows arbitrary configuration changes. Like the

single-server version, a new configuration is proposed as a special command, but instead of immediately switching to the new configuration, the replicas transition to an intermediate "joint configuration" consisting of both the old and new members. In this state, elections and commit operations require support from majorities of *both* configurations (not their union) to succeed. Once a command is committed under the joint configuration, it is safe to transition to the new configuration.

$$
\begin{aligned}
\mathit{Config} &\triangleq \mathit{Set}(\mathbb{N}_{\mathit{nid}}) * \mathit{Option}(\mathit{Set}(\mathbb{N}_{\mathit{nid}})) \\
\mathsf{R1}^{+}(C, C') &\triangleq \exists\, \mathit{old}.\, (C = (\mathit{old}, \bot) \land C' = (\mathit{old}, \_)) \lor \\
&\quad\ \exists\, \mathit{new}.\, (C = (\_, \mathit{new}) \land C' = (\mathit{new}, \bot)) \\
\mathit{isQuorum}(S, (\mathit{old}, \mathit{new})) &\triangleq |\mathit{old}| < 2 * |S \cap \mathit{old}| \land \\
&\quad\ (\mathit{new} = \bot \lor |\mathit{new}| < 2 * |S \cap \mathit{new}|)
\end{aligned}
$$

The essential point here is that the joint configuration requires majorities from both configurations. This ensures that, when transitioning from the old to joint configuration or joint to new, there exists a majority of supporters in both sets, which guarantees that they have a replica in common.

Primary Backup Instead of relying on majorities, another approach is something similar to a primary backup protocol, such as Chain Replication [van Renesse and Schneider 2004], in which one replica or set of replicas (the *primary*) is responsible for committing commands, while the others serve as passive backups. A quorum is then any set containing the primary, which means the set of passive backups can change arbitrarily.

$$
\begin{aligned}
\mathit{Config} &\triangleq \mathbb{N}_{\mathit{nid}} * \mathit{Set}(\mathbb{N}_{\mathit{nid}}) \\
\mathsf{R1}^{+}((P, \_), (P', \_)) &\triangleq P = P' \\
\mathit{isQuorum}(S, (P, \_)) &\triangleq P \in S
\end{aligned}
$$

In this case, the primary is constant and is a member of every quorum, so all quorums

obviously intersect. The limitation of this is if the primary crashes then all progress is blocked. A more reliable alternative is to use one of the previous approaches to manage a set of primaries that can be replaced as needed. For example, using Raft's single-server scheme, one could have a set of three primaries and define a quorum as any set containing two of them. Primaries can then be replaced one at a time, and passive backups can still be freely added or removed. This also allows replicas to move between the primary and passive sets to dynamically adjust to varying availability needs.

Dynamic Quorum Sizes With a set of n replicas, a quorum size of $\lceil n/2 \rceil$ allows only one replica to be added or removed at a time while still guaranteeing overlap. On the other hand, a quorum size of n means $n - 1$ replicas can safely be changed at once. In general, larger quorums allow for faster reconfiguration, but are less fault tolerant. This type of trade-off is why protocols like Vertical Paxos [Lamport et al. 2009] allow quorum size to be adjusted dynamically. ADORE's reconfiguration scheme supports this by adding quorum size (q) to the configuration.

$$
\begin{aligned}
\mathit{Config} &\triangleq \mathbb{N} * \mathit{Set}(\mathbb{N}_{nid}) \\
\mathsf{R1}^{+}((q, C), (q', C')) &\triangleq (C \subseteq C' \wedge |C'| < q + q') \vee \\
&\quad\ (C' \subseteq C \wedge |C| < q + q') \\
\mathit{isQuorum}(S, (q, C)) &\triangleq q \leq |S \cap C|
\end{aligned}
$$

The argument for why this guarantees overlap is a generalization of the single-server case. Intuitively, if the sum of the quorum sizes is greater than the size of the larger sets, then two quorums together contain at least that many elements. Then, by the pigeonhole principle, at least one element is a duplicate, so the quorums must have it in common.

5.7 Evaluation and Discussion

Proof Effort and Experience The total amount of Coq to implement and prove the safety of Adore is approximately 10.8k lines.[1] Of that, 2.3k are generic well-formedness invariants about the tree data structure (e.g., proving the absence of cycles), and 4k are part of a general library of utility functions and lemmas, leaving 4.5k for the kind of proof shown in Section 5.4. We also performed the same safety proof using the CADO model (Adore without reconfiguration), which took approximately 1.3k lines (excluding the tree properties). The CADO proof took one person approximately two weeks to complete and adding reconfiguration took another three.

For comparison, the safety proof of the network-based, non-reconfigurable Paxos specification used in the Advert refinement proof (Section 4.5) required approximately 5k lines. The relative ease with which Adore handles a much more complicated problem is a strong demonstration that an abstraction designed specifically for protocol-level reasoning is a powerful tool.

The primary features of Adore that make it ideal for protocol-level reasoning are its atomic interface and cache tree. Reducing the complexities of network communication to four simple operations greatly reduces the number of cases to consider. The cache tree is also an expressive abstraction that captures important information from log-based models, but makes explicit certain invariants, such as the existence of a common prefix of committed commands.

[1] Despite the conceptual similarities with Advert, almost none of the code is shared. Instead, we took the opportunity to reimplement the cache tree and other data structures in a style that is easier to use in Coq.

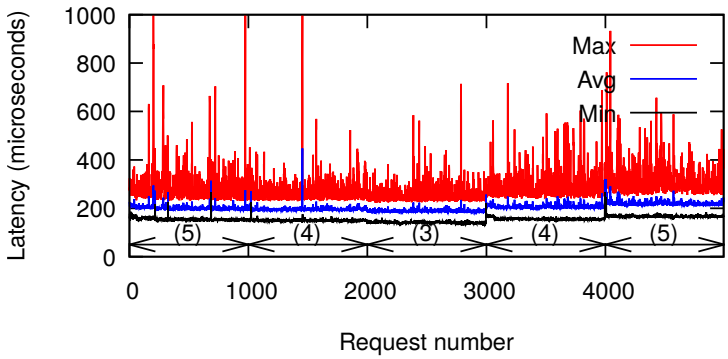


Figure 5.13: OCaml Raft performance with reconfiguration. (n) indicates the number of replicas.

Refinement The refinement proof between the network-based model of a Raft-like protocol and Adore took approximately nine weeks and 13.8k lines of Coq, of which 2.5k is the refinement between SRaft and Adore. The protocol is parameterized by the same *isQuorum* and R1$^+$ predicates as Adore, which means the refinement proof actually holds for a large family of protocols with different reconfiguration schemes. Instantiating these parameters and proving that they satisfy the necessary properties is trivial. Adore's codebase includes six examples (the four from Section 5.6 and two others) that took only a few days and about 200 lines in total for both the definitions and proofs (several rely on the proof that majority subsets overlap, which is an additional 100 lines).

OCaml Extraction and Performance Using Coq's support for extraction to OCaml we created an executable version of the Raft network-based specification and, with a small, unverified network library wrapper, evaluated its performance on Amazon EC2 with `m4.xlarge` instances. The experiment adds or removes a replica after every 1000 client requests, starting with five replicas, dropping to three, then increasing back to five. Figure 5.13 shows the maximum, mean, and minimum latencies for processing each client

command over eight runs.

Reconfiguration adds a small delay, especially when the number of replicas increases, but it is within the normal range of sporadic latency spikes. Our aim is not to make any strong performance claims, but merely to demonstrate that Adore's safety guarantees can extend to verified executable code (excluding the OCaml extraction process, compiler, and network libraries) without being unreasonably slow.

5.8 Summary

Existing models for distributed systems are not ideal for protocol-level verification because they mix concerns from different abstraction levels, which makes it difficult to reason about important but complex operations like reconfiguration. Adore demonstrates that the ADO model works very well in this setting by hiding irrelevant network details and emphasizing the inherent tree-like structure of the global system state due to partial failures. These advantages enabled us to complete the first mechanized safety proof for a generic consensus protocol with a hot reconfiguration algorithm.

Adore shows that the core ADO model can be extended with additional operations without significantly complicating the safety proof. In Chapter 6 we will consider a different kind of extension and show how to prove liveness as well as safety.

Chapter 6

AdoB: Atomic Distributed Objects for Benign and Byzantine Consensus

This chapter presents a third version of the ADO model, AdoB. Like Advert and Adore, AdoB is a protocol-level abstraction, but it expands the scope from benign consensus safety properties to the safety and liveness of both benign and byzantine failure models. Section 6.1 motivates the need to verify these properties and Section 6.2 summarizes AdoB's approach. The formal details are divided into two sections. First, Section 6.3 presents a benign version of the model along with safety and liveness proofs, then Section 6.4 shows how these are generalized to also support the byzantine case. Section 6.5 explains the primary steps of a refinement with a network-level specification, and Section 6.6 discusses some important lessons learned. Finally, Section 6.7 summarizes the results.

6.1 Motivation

As discussed in Section 2.1.2, in order to handle byzantine failures, consensus protocols require more communication than their benign counterparts. For example, because a leader cannot be trusted to propose the same method to all replicas, a pre-commit phase is required, which constructs a certificate that can be included with a commit request as evidence of its safety. This, and other differences, may seem to imply that benign and byzantine consensus are fundamentally different, and that a single abstraction cannot hope to describe both.

Upon closer inspection, however, these differences are not so deep. The goal of the pre-commit phase is to ensure that some trusted replica has vouched for both the leader's legitimacy and the safety of the proposed method. In the byzantine case, this trust is achieved through acquiring a sufficiently large quorum of votes, while, in the benign case, the leader itself acts as the trusted party. We can see, therefore, that the benign local update phase serves the same purpose as the byzantine pre-commit, and everything else is merely an implementation detail.

AdoB formalizes this relation by precisely identifying the few key differences between benign and byzantine consensus and abstracting over them in order to obtain a unified model. This allows one to prove a safety property in a very similar style to Adore, but AdoB goes further and also supports reasoning about liveness. This is a very important property in practice because safety only concerns what *may not* happen, whereas liveness is about what *must* happen.

The biggest challenge in proving liveness is deciding how to correctly model timeouts.

These are different from other operations because they use an all-to-all communication pattern instead of relying on a leader to coordinate, and it is not obvious how to model this as an atomic event. In fact, several of our early attempts had subtle mistakes that made it impossible to refine AdoB with a network-based specification. We discuss these bugs further in Section 6.6.1.

6.2 Overview

Adore's version of the cache tree (Section 5.3) has proven to be very useful for proving high-level properties of consensus protocols; however, it is lacking in two areas for our goals. Firstly, it has no concept of time or timeouts, so it cannot be used to prove liveness. Secondly, it assumes a benign setting and has no way to model byzantine behaviors. AdoB makes several minor modifications to address these shortcomings; however, for simplicity, it omits the `reconfig` operation.

Timeouts The first problem is addressed by adding a new type of timeout cache (*TCache*) and adjusting `pull`, `invoke`, and `push` to either succeed (creating an *ECache*, *MCache*, or *CCache*, respectively), or fail with a *TCache*. Though this appears at first to be a relatively straightforward addition, we found it to be a particularly 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.

Byzantine Replicas 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.

As a result of these weakened conditions, a quorum of votes is no longer sufficient to guarantee the existence of a common honest voter for consecutive elections and commits. Instead, all operations require a super quorum, which is typically a 2/3 majority but can be defined other ways as well. To remain as general as possible, as in Adore, these are represented by an *isSQuorum* parameter.

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`. In fact, the new byzantine `invoke` is nearly identical to `push`.

Merging the Models The final key to merging the benign-only and byzantine-only versions of AdoB is to observe 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 be 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

Parameters

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

Assumptions

$$\begin{array}{ll} (\textsc{Disjoint}) & nonfaulty \cap faulty = \emptyset \\ (\textsc{Overlap}) & isQuorum(Q) \wedge isQuorum(Q') \implies Q \cap Q' \neq \emptyset \end{array}$$

Figure 6.1: Benign AdoB configuration and quorum parameters and assumptions.

a special parameterized *method quorum* (*mquorum*), we can cover both cases in a single specification that can be instantiated in different ways.

6.3 AdoB for Benign Consensus

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 byzantine failures, there are several key design decisions that enable a smooth transition to the generalized case in Section 6.4.

6.3.1 Semantics

State Figure 6.2 defines the system state (Σ) as a pair of a cache tree, and every replica's local logical timestamp. 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 assumed to be *honest* (Figure 6.1). The definition of a quorum is flexible, but it must at least satisfy the property that any two quorums have a non-empty

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

Figure 6.2: Benign AdoB state definitions.

intersection (Overlap). This makes AdoB more generic than if quorums were fixed as simple majorities. 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 (*Commit*), 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*.

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 ($>$) by comparing timestamps and using *cRank* as a tie-breaker, which is chosen to preserve the invariant that a cache is greater (with respect to $>$) than its ancestors. Figure 6.4 defines $>$ along with other useful functions on caches and cache trees.

AdoB draws a similar distinction between *voters* and *supporters* as Adore. A replica's "local state" is represented by its *active* cache. Some operations update a replica's active

$$
\begin{aligned}
Op \triangleq\ & \texttt{pull} : \mathbb{N}_{nid} \to \Sigma \to \Sigma \\
\mid\ & \texttt{invoke} : \mathbb{N}_{nid} \to \mathit{Method} \to \Sigma \to \Sigma \\
\mid\ & \texttt{push} : \mathbb{N}_{nid} \to \Sigma \to \Sigma
\end{aligned}
$$

Figure 6.3: Benign AdoB operations.

cache, but a replica may also only witness and approve the creation of a cache without changing its own. The most recent such cache is called its *voted* cache. A replica's active cache 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` (Figure 6.3). Each takes its caller's node ID and the current state and returns a new state. The `invoke` operation additionally 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 represent with a logical oracle ($\mathbb{O}$). The oracle's responsibility is to abstract over the many ways messages may interleave or fail and return a simple success (*Ok*) or timeout (*Timeout*) result. In Figure 6.5, we use the notation $\mathbb{O} \vdash op : st \rightsquigarrow st'$ to represent operation op called on state st with oracle $\mathbb{O}$ results in st'.

Pull The `pull` operation models an election by asking $\mathbb{O}_{pull}$ (Figure 6.6) to choose a set of voters (Q), a sufficiently up-to-date cache (C_{max}), and the next timestamp (t). It then

$$
\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) \\
&\quad \vee (time(C_1) = time(C_2) \wedge cRank(C_1) > cRank(C_2)) \\
C_1 \succeq C_2 &\triangleq (time(C_1), cRank(C_1)) = (time(C_2), cRank(C_2)) \vee C_1 \succ C_2 \\
voters(C) &\triangleq \textbf{if } C = ECache(\_, \_, Q) \textbf{ then } Q \textbf{ else} \\
&\quad \textbf{if } C = MCache(\_, \_, Q, \_) \textbf{ then } Q \textbf{ else} \\
&\quad \textbf{if } C = CCache(\_, \_, Q) \textbf{ then } Q \textbf{ else} \\
&\quad \textbf{if } C = TCache(\_, Q, \_) \textbf{ then } Q \\
supporters(C) &\triangleq \textbf{if } C = ECache(nid, \_, \_) \textbf{ then } \{nid\} \textbf{ else} \\
&\quad \textbf{if } C = MCache(nid, \_, \_, \_) \textbf{ then } \{nid\} \textbf{ else} \\
&\quad \textbf{if } C = CCache(\_, \_, Q) \textbf{ then } Q \textbf{ else} \\
&\quad \textbf{if } C = TCache(\_, \_, Q) \textbf{ then } Q \\
freshCID(tr) &\triangleq \max\{cid(C) \mid C \in tr\} + 1 \\
addLeaf(st, C_P, C_{new}) &\triangleq (tree(st)[freshCID(tree(st)) \mapsto (C_P, C_{new})], times(st)) \\
setTimes(st, Q, t) &\triangleq (tree(st), times(st)[s \mapsto t \mid \forall s \in Q \cap honest]) \\
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)\} \\
activeC(tr, s) &\triangleq \max_{\succ}\{C \in tr \mid s \in supporters(C) \wedge C = CCache(\_)\} \\
canElect(tr, C, Q) &\triangleq (C = CCache(\_) \vee C = TCache(\_)) \\
&\quad \wedge \forall s \in Q \cap honest.\, C \succeq active(tr, s) \\
canInvoke(tr, C, nid, Q) &\triangleq C = ECache(nid, \_, \_) \\
&\quad \wedge \forall s \in Q \cap honest.\, C \succeq voted(tr, s) \\
canCommit(tr, C, nid, Q) &\triangleq C = MCache(nid, \_, \_, \_) \\
&\quad \wedge \forall s \in Q \cap honest.\, C \succeq voted(tr, s) \\
canTimeout(tr, C, Q) &\triangleq \forall s \in Q \cap honest.\, C \succeq activeC(tr, s)
\end{aligned}
$$

Figure 6.4: Benign AdoB auxiliary definitions.

PullOk

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

InvokeOk

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

PushOk

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

Timeout

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

Figure 6.5: 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.

updates the voter's timestamps with *setTimes* to reflect their vote, and adds a new *ECache* child to C_{max}. This represents a logical marker that at this point, C_{max} is the most recent cache among this quorum of voters.

$\mathbb{O}_{pull}$ 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

$$
\begin{aligned}
\mathit{TimeoutResult} &\triangleq \mathit{Timeout}(\mathit{Set}(\mathbb{N}_{nid}) * \mathit{Set}(\mathbb{N}_{nid}) * \mathit{Cache} * \mathbb{N}_{time}) \\
\mathbb{O}_{pull} &: \Sigma \to \mathbb{N}_{nid} \to (\mathit{Ok}(\mathit{Set}(\mathbb{N}_{nid}) * \mathit{Cache} * \mathbb{N}_{time}) \mid \mathit{TimeoutResult}) \\
\mathbb{O}_{invoke} &: \Sigma \to \mathbb{N}_{nid} \to (\mathit{Ok}(\mathit{Cache}) \mid \mathit{TimeoutResult}) \\
\mathbb{O}_{push} &: \Sigma \to \mathbb{N}_{nid} \to (\mathit{Ok}(\mathit{Set}(\mathbb{N}_{nid}) * \mathit{Cache}) \mid \mathit{TimeoutResult})
\end{aligned}
$$

ValidPullOracleOk

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

ValidInvokeOracleOk

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

ValidPushOracleOk

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

ValidOracleTimeout

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

Figure 6.6: Valid benign AdoB oracle conditions. The conditions for timing out are identical regardless of the operation so ValidOracleTimeout is parameterized by *op*.

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.

Note that, in several places, only the honest voters are considered by intersecting Q with *honest* (e.g., *setTimes*, *canElect*). This is redundant because all voters are currently

honest; however, expressing it this way brings this model more in line with the byzantine case and will make it simpler to generalize later.

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 to do so. This has important implications for future `pull` operations because it affects the valid choices of C_{max}. Recall that *canElect* requires that C_{max} is at least as recent as its voters' active (i.e., supported) caches. This set of voters constitutes a quorum, which means at least one must also be a supporter of the *CCache*. Therefore, the next election is guaranteed to be "aware of" 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 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 new *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. Rather than being initiated by a leader and supported by a quorum of replicas, a timeout is a collective action by a quorum of replicas without a leader's involvement. 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. These sets may or may not overlap.

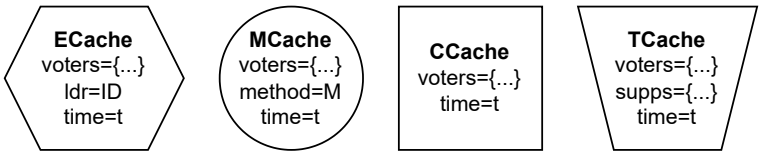
Included in each timeout message from Q_{vote} is the replica's active cache. These are collected and forwarded to the leader of the next round to prompt it to begin an election. The oracle enforces this with *canTimeout*, which confirms C_{max} is at least as recent as the voters' latest supported *CCache* (*activeC*). 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, which is necessary to ensure that the system progresses through rounds sequentially.

Though these rules seem reasonable, it is not obvious whether some slight modifications might not be equally valid. For example, what happens if one requires $C = activeC(tr, s)$ in *canTimeout*, or drops Q_{supp} from ValidOracleTimeout and uses Q_{vote} for both voters and supporters? These alternatives may seem reasonable, indeed one can even still prove safety and liveness, but in fact they are invalid because they do not faithfully model the

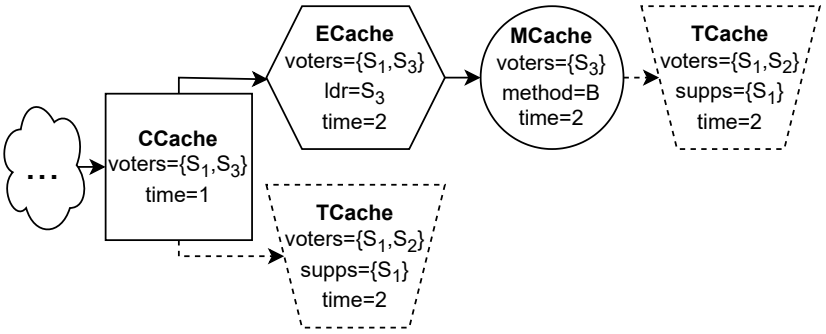
behaviors of real protocols. The fact that these minor alterations can render the abstraction invalid demonstrates that correctly modeling timeouts in AdoB is a subtle challenge, and that a refinement proof from a lower-level model is essential to have faith in a high-level model. Section 6.6.1 discusses these issues further.

Example For the most part, `pull`, `invoke`, and `push` behave similarly to how they were described in Figures 3.2 and 3.3. In the steady state, branches still 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 Figure 3.3b; 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 Figure 3.3b is now disallowed.

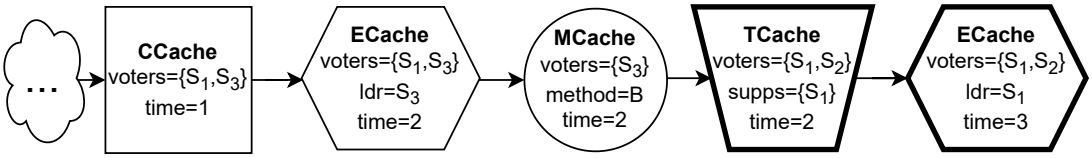
Instead, before creating an *ECache* for time 3, there must first be a *TCache* for time 2. In Figure 6.7 the three valid options for the *TCache*'s parent (caches that satisfy *canTimeout*) are an uncommitted *MCache*, its parent *ECache*, and the latest *CCache*. If the *MCache* is chosen, then the next leader picks up where the previous one left off and continues extending the same branch. Otherwise, if the *CCache* is chosen, then a fork is created and the *MCache* is abandoned. 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*.



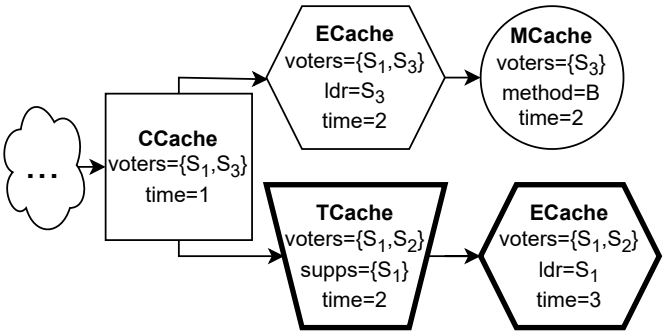
(a) The types of caches. *ECaches*, *MCaches*, and *CCaches* are the same as Figure 5.2 except *MCaches* now also have a set of voters. *TCaches* represent timeouts.



(b) S_3 times out while committing. There are three possible locations for *TCache* (the *ECache* option is not shown because choosing it is effectively the same as choosing the *CCache*).



(c) Option 1: The next leader continues building off the previous *MCache*.



(d) Option 2: The next leader starts a new branch.

Figure 6.7: An example of a timeout in AdoB.

6.3.2 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 C.3 and the full proofs can be found in the supplementary materials.

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

Theorem 3 (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 supposing neither cache is the other's descendant and deriving 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 every *ECache* must be a descendant of every earlier *CCache*.

Lemma 4 (Election Follows Commit)**.** *For any CCache C and ECache C', if $C' \succ C$ 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. It is an intuitively simple property that leaders cannot be elected if they are missing any committed methods. In AdoB it is equally simple to express 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 Figure 6.1). This is easily instantiated by the typical $f + 1$ out of $2f + 1$ majority, but it also allows for more exotic variants where, for instance, replicas have weights and a quorum is any set whose total weight exceeds some threshold (similar to the proof-of-stake scheme [Saleh 2020] used by some blockchain protocols).

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 6 (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, as it could just call `pull` forever and nothing would ever be committed. Instead, we must assume 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 (e.g., `pull` because it has completed its previous round), the strategy will decide to call that operation within some finite number of steps, and furthermore, the replica will not participate in any other operations before that point.

Definition 7 (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 4 (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 that an operation succeeds. It is entirely possible for a leader to believe it is eligible to call `push` but be preempted by another leader's `pull`. As long as the attempt is made, we can assume that under reasonable conditions, it will eventually have an opportunity to succeed.

In particular, we assume that the network is partially synchronous [Dwork et al. 1988] in order to rule out pathological cases, such as every message being dropped. Recall from Section 2.1.3 that this means that, after some global stabilization time (GST), messages between non-faulty replicas are delivered in finite time. We express this through assumptions that, after GST, all non-faulty replicas will vote for any valid `pull` or `push` request.

Definition 8 (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 $\mathbb{O}_{push}$.*

This definition requires that, once GST is reached, the network continues to behave synchronously forever. This is slightly stronger than necessary as we actually only require synchrony to hold for one complete round (the time to complete a `pull`, `invoke`, and `push`). However, expressing this more precise bound complicates the formalization, so it is common to make this simplifying assumption [Bravo et al. 2020; Hawblitzel et al. 2015a; Losa and Dodds 2020].

The final necessary assumption is that eventually a non-faulty leader has the opportunity to be elected. HotStuff and Jolteon guarantee this with a round-robin rotating leadership, but other schemes such as Raft's randomized election timeouts are also possible. AdoB therefore simply assumes the existence of an arbitrary deterministic order that eventually selects a non-faulty replica.

Definition 9 (Fair Rotating Leadership)**.** *Leaders are determined for each round according to some deterministic schedule. The order may be completely arbitrary except that a non-faulty replica must always be selected within a finite number of rounds.*

Armed with these assumptions, the liveness proof decomposes into two main steps: showing that the system is always able to progress to the next round by either committing a method or timing out; and, after GST, a non-faulty leader is eventually reached who will commit a method. The first part relies on a notion of *global time* that indicates the latest

round in which any replica has participated.

Definition 10 (Global Time)**.** *The global time of a cache tree is the timestamp of the most recent (with respect to* $>$*) ECache or TCache.*

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

Lemma 5 (Global Time Advances)**.** *Given any cache tree and productive strategy, a new cache tree is eventually produced with a strictly larger global time than the original tree.*

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, a *TCache* on timeout.

Now, thanks to Lemma 5 and Definition 9, we know that, if one waits long enough, a round will begin with a non-faulty leader. Then, because we have reached GST, Definition 8 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 the pre-existing custom library of general lemmas and tactics from Adore, 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.

The safety proof very closely follows the high-level structure of Adore's safety proof. In fact, several of the top-most theorems are exactly the same. The differences are primarily localized to lower-level lemmas that perform case analysis on the different cache types (e.g., the *TCache*, which Adore lacks). This contributed to the faster proof times and is a strong demonstration of the benefits provided by the ADO model's isolation of protocol-level behaviors from irrelevant network-level details.

6.4 AdoB for Generalized Consensus

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

6.4.1 Adapting to Byzantine Consensus

Thanks to our efforts in Section 6.3 to bring out the shared structure of the benign and byzantine cases, only three additional changes are required to support byzantine consensus. Figures 6.8 to 6.10 highlight these modifications with 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 do 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

Parameters

$$\begin{array}{rcl@{\qquad}rcl} \boxed{honest} & : & Set(\mathbb{N}_{nid}) & isQuorum & : & Set(\mathbb{N}_{nid}) \to \mathbb{B} \\ \boxed{byzantine} & : & Set(\mathbb{N}_{nid}) & \boxed{isSQuorum} & : & Set(\mathbb{N}_{nid}) \to \mathbb{B} \\ conf & \triangleq & \boxed{honest} \cup \boxed{byzantine} & leaderAt & : & \mathbb{N}_{time} \to \mathbb{N}_{nid} \end{array}$$

Assumptions

$$\begin{array}{rl} \text{(Disjoint)} & \boxed{honest} \cap \boxed{byzantine} = \emptyset \\ \text{(Overlap)} & isQuorum(Q) \wedge isQuorum(Q') \implies Q \cap Q' \neq \emptyset \\ (\boxed{\text{SOverlap}}) & isSQuorum(Q) \wedge isSQuorum(Q') \implies Q \cap Q' \cap honest \neq \emptyset \end{array}$$

Figure 6.8: Byzantine AdoB configuration and quorum parameters and assumptions. The replicas are no longer all honest. Super quorums must have an honest overlap.

InvokeOk

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

All other rules are the same as in Figure 6.5.

Figure 6.9: 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.

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 Figure 6.8). As with regular quorums, we do not fix super quorums to any 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 Figure 6.10.

Note that, while the model separates honest and byzantine replicas, it is important that

$$\boxed{\mathbb{O}_{invoke}} : \Sigma \to \mathbb{N}_{nid} \to (Ok(\boxed{Set(\mathbb{N}_{nid})} * Cache) \mid TimeoutResult)$$

ValidPullOracleOk

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

ValidInvokeOracleOk

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

ValidPushOracleOk

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

ValidOracleTimeout

$$\frac{\begin{array}{c} \boxed{isSQuorum(Q_{vote})} \qquad Q_{supp} \cap honest \neq \emptyset \\ canTimeout(tree(st), C_{max}, Q) \qquad \forall s \in (Q_{vote} \cup Q_{supp}) \cap honest.\ times(st)[s] \leq t \\ \exists s \in Q_{vote} \cap honest.\ times(st)[s] = t \end{array}}{\mathbb{O}(st, nid) = Timeout(Q_{vote}, Q_{supp}, C_{max}, t)}$$

Figure 6.10: Valid byzantine AdoB oracle conditions. Quorums are replaced by super quorums.

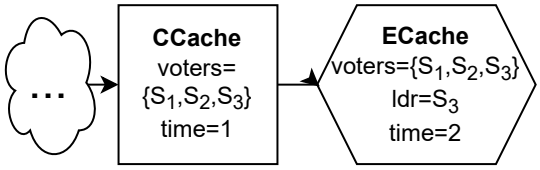
we never rely on this knowledge to determine an operation's outcome. That is why *honest* is only used to weaken preconditions (e.g., $\forall s \in Q \cap honest.\ P(s)$ exempts byzantine replicas from satisfying P), or when a group is large enough to draw conclusions about its members (SOverlap can safely assume a common honest replica exists due to the properties of *isSQuorum*). In Section 6.5, 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,

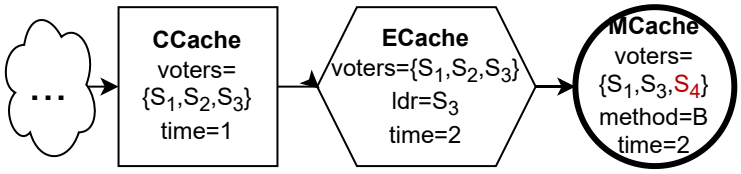
could attempt to trick other replicas into committing invalid states either by proposing an out-of-date cache, or by equivocating and proposing different caches to different replicas. To rule out this possibility, leaders must gather evidence that at least a super quorum has approved a proposed cache before it can be committed. Previously, this evidence was provided implicitly by `invoke` as it meant the leader unilaterally gave its approval for an *MCache*, which is enough when everyone behaves honestly. Now, `invoke` must actually gather a super quorum of voters, which is decided by $\mathbb{O}_{invoke}$ (Figure 6.10). The preconditions are the same as before but extended to every replica in Q instead of just the leader. Intuitively, it is as if every voter performs a local dry run of its own benign `invoke`.

One may wonder if the four outcomes in Figure 6.10 really capture all possible behaviors of a malicious replica. What happens, for instance, if a byzantine leader colludes with other byzantine replicas to update their local logs in invalid ways? We make the design decision to model such operations as having no effect in AdoB. We then justify this decision by proving that a specification that allows these behaviors refines AdoB (see Section 6.5.1). The advantage of this approach is it allows us to handle this type of implementation-level complication independently from general consensus properties such as safety and liveness, which disentangles and greatly simplifies the proofs.

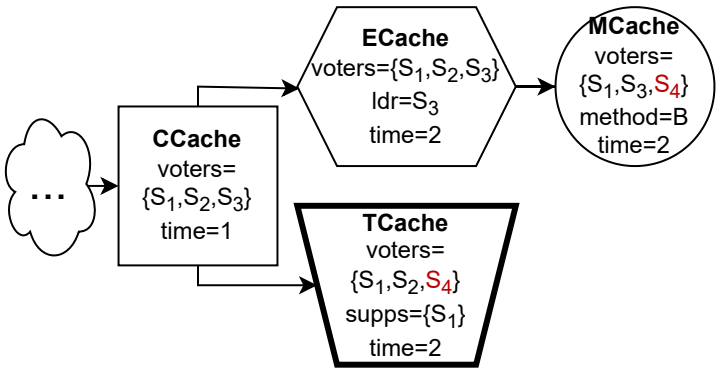
Examples The addition of byzantine replicas does not drastically change the behaviors allowed by AdoB. Figure 6.11 shows a typical cache tree with one byzantine replica (S_4, shown in red) and three honest replicas (S_1, S_2, S_3). In Figure 6.11b, the leader, S_3, successfully invokes a method. The main difference from the benign setting is it requires a super quorum of votes (at least 3 out of 4). This ensures that, although one of the voters is



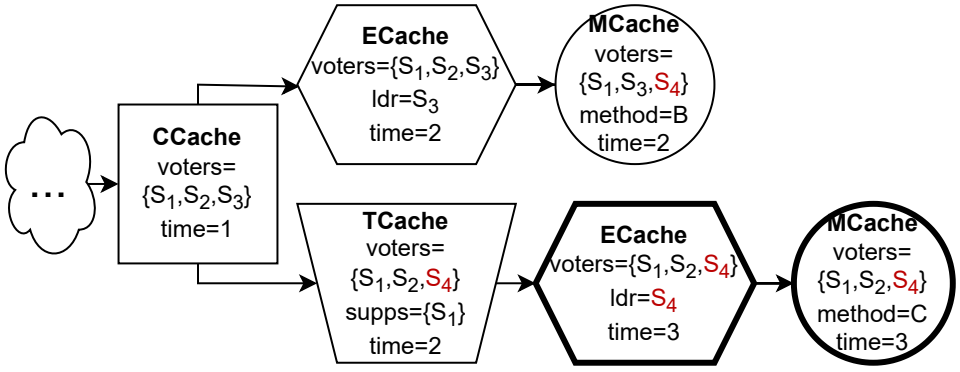
(a) S_3 is the leader. The configuration is S_1, S_2, S_3, S_4 and all but S_4 are honest.



(b) S_3 invokes a method with votes from a super quorum, including S_4.



(c) S_1, S_2, S_4 time out waiting for S_3 to commit. S_4 could be lying about its time, but this is still safe because S_1 and S_2 form an honest quorum.



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

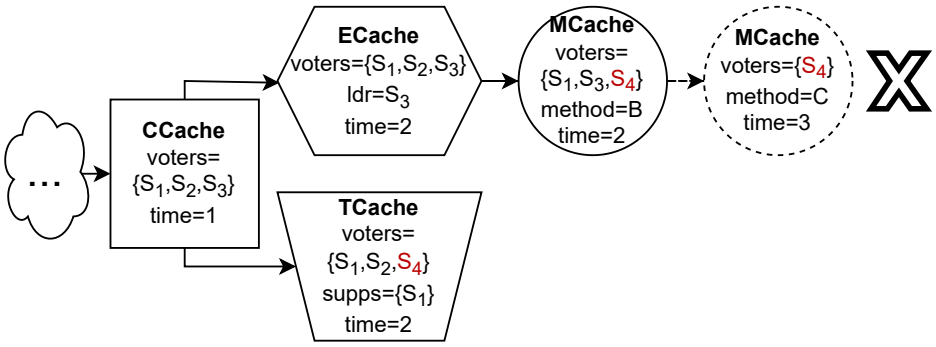
Figure 6.11: Allowed behaviors in byzantine AdoB.

S_4 and cannot be trusted, 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.

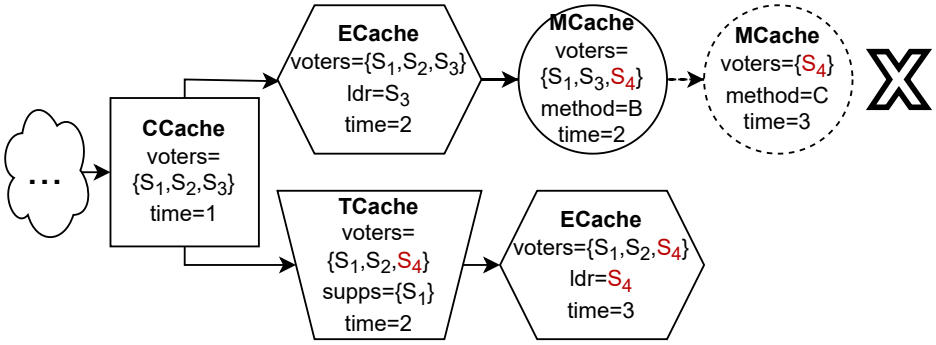
In Figure 6.11c, S_1, S_2, and S_4 time out while waiting for S_3 to commit and create a *TCache*. It is possible that S_4 is lying about its timer running out, but, once again, the existence of a super quorum of voters ensures the *TCache* is safe despite a potentially malicious participant. Finally, in Figure 6.11d, S_4 is successfully elected and invokes a method. This shows that byzantine replicas do not necessarily always act maliciously, and, as long as they behave honestly, they can contribute to the committed state.

Figure 6.12 shows that, even when byzantine replicas do act maliciously, they are limited in the damage they can cause. For example, S_4 could never create the *MCache* with the dotted outline in Figure 6.12a 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 Figure 6.12b, 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*.

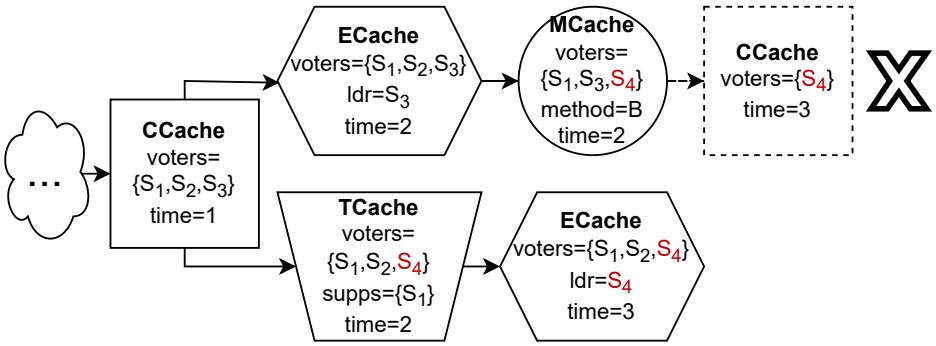
For the same reasons, S_4 also cannot commit a method from a previous round (Figure 6.12c). The *TCache* is more recent than the *MCache* for method *B*, so S_4 can never acquire enough votes. Nor can it create a *CCache* on its own branch without first invoking a method (Figure 6.12d). Replicas require 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*.



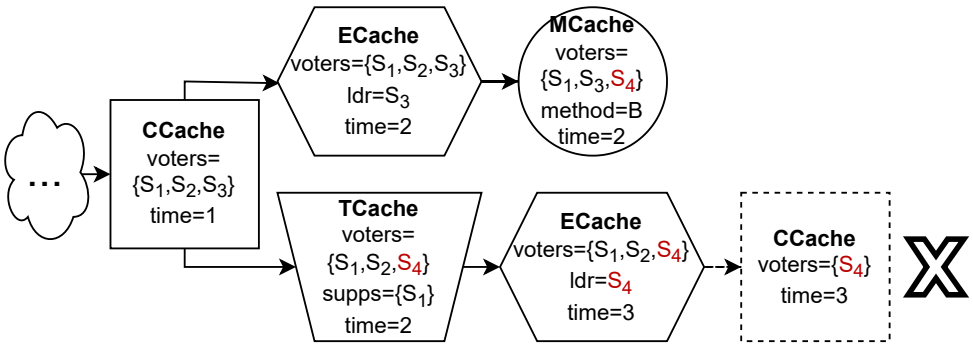
(a) S_4 cannot invoke a method without being elected.



(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.

Figure 6.12: Disallowed behaviors in byzantine AdoB. Dotted outlines represent impossible cases.

6.4.2 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 that they can therefore be totally ordered. The byzantine case guarantees this by requiring a super quorum of voters for every operation, and the benign case requires quorums for `pull`, `push`, and timeouts, but, at first glance, 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 Figure 6.13) that can be

Parameters

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

Assumptions

$$\begin{aligned}
(\boxed{\text{MOverlap}})\ & isMQuorum(ldr, Q) \land isMQuorum(ldr, Q') \\
& \implies Q \cap Q' \cap honest \neq \emptyset \\
(\boxed{\text{MSOverlap}})\ & isMQuorum(ldr, Q) \land isSQuorum(Q') \land ldr \in Q' \\
& \implies Q \cap Q' \cap honest \neq \emptyset
\end{aligned}$$

Figure 6.13: Method quorum (*mquorum*) parameters and assumptions. Any two *mquorums* for the same leader must have a common honest voter. Any *mquorum* and super quorum that includes the same leader must have a common honest voter.

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

Figure 6.14: $\mathbb{O}_{invoke}$ replaces super quorums with *mquorums*.

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 (Figure 6.14).

It is not difficult to instantiate *isMQuorum* and prove it satisfies the assumptions for the benign and byzantine cases, as Figure 6.15 demonstrates. For the benign case, MOverlap holds trivially because we know $ldr \in Q$ and $ldr \in Q'$ from the definition of *isMQuorum* and $ldr \in honest$ because $byzantine = \emptyset$. MSOverlap follows the same reasoning because

Benign

$$
\begin{aligned}
byzantine &\triangleq \emptyset \\
isQuorum(Q) &\triangleq |Q| > |conf|/2 \\
isSQuorum(Q) &\triangleq isQuorum(Q) \\
isMQuorum(ldr, Q) &\triangleq ldr \in Q
\end{aligned}
$$

Byzantine

$$
\begin{aligned}
byzantine &: Set(\mathbb{N}_{nid}) \\
isQuorum(Q) &\triangleq |Q| > |conf|/2 \\
isSQuorum(Q) &\triangleq |Q| > 2|conf|/3 \\
isMQuorum(ldr, Q) &\triangleq isSQuorum(Q)
\end{aligned}
$$

Figure 6.15: Quorum instantiations for benign and byzantine settings.

$ldr \in Q'$ by assumption. In the byzantine case, MOverlap and MSOverlap both follow from SOverlap and Overlap.

6.4.3 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.

Just as in the benign case everything rested on the existence of overlapping quorums, the generalized benign/byzantine case relies on an honest overlap 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 6 (Push Max Parent)**.** *If $\mathbb{O}_{push}$ returns Ok for some replica, then the cache it selects is as least as recent (according to $\succeq$) 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 $C_M \succeq C$. 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 $C_M \succeq C$.

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. The few remaining steps to generalize it only require minor, local changes to the proofs.

6.5 Refinement

The safety and liveness of AdoB is only meaningful if it faithfully models the behavior of actual benign and byzantine consensus protocols. We demonstrate that it does by proving that a network-based specification of a novel Jolteon variant refines AdoB. We call this variant GenJolteon because it is capable of tolerating either benign or byzantine faults depending on the instantiation of *mquorum*. As usual, we specify it in terms of a network-based model; i.e., a set of replicas with local logs communicating over a network. To lessen the gap between this model and AdoB, we make certain assumptions of the network to make it behave more synchronously. These assumptions can then be relaxed in a series of additional refinements that can be transitively linked together.

In this section, we focus on the uppermost layer that connects the simplified network model to AdoB, which we have proved in Coq. This is by far the most interesting and challenging step because the significant gap between the abstraction levels leaves the most room for specification bugs. At the time of writing, we have completed manual proof sketches for the layers that relax the network assumptions, and have begun the Coq formalizations. We do not anticipate significant conceptual challenges in these steps as they are very similar to the network reordering logic used in Adore (Section 5.5) and in prior work [Chajed et al. 2018; Hawblitzel et al. 2015a; v. Gleissenthall et al. 2019].

6.5.1 Network-Based Specification

The global state of the abstract network-based model (Figure 6.16) consists of a set of local states and a network, which is represented as a pair of bags of sent and delivered messages.

$$
\begin{aligned}
\Sigma_{\text{net}} &\triangleq (\mathbb{N}_{nid} \rightharpoonup \mathit{Replica}) * \mathit{Network} \\
\mathit{Replica} &\triangleq \mathbb{N}_{time} * \mathit{Log} * \mathit{Phase} \\
\mathit{Network} &\triangleq \mathit{Set}(\mathit{Msg}) * \mathit{Set}(\mathit{Msg}) \\
\mathit{Log} &\triangleq \mathit{List}(\mathbb{N}_{time} * \mathit{Set}(\mathbb{N}_{nid}) * \mathit{Method}) \\
\mathit{Phase} &\triangleq \mathit{Idle} \mid \mathit{VotedInvoke} \mid \mathit{VotedCommit} \\
&\quad \mid \mathit{Elected} \mid \mathit{Invoked} \mid \mathit{WaitForAcks} \mid \ldots \\
\mathit{Msg} &\triangleq \mathit{Request}(\mathbb{N}_{nid} * \mathbb{N}_{nid} * \mathbb{N}_{time} * \mathit{Cmd}) \\
&\quad \mid \mathit{RequestMany}(\mathbb{N}_{nid} * \mathit{Set}(\mathbb{N}_{nid}) * \mathbb{N}_{time} * \mathit{Cmd}) \\
&\quad \mid \mathit{Ack}(\mathit{Set}(\mathbb{N}_{nid}) * \mathbb{N}_{nid} * \mathbb{N}_{time} * \mathit{Cmd}) \\
\mathit{Cmd} &\triangleq \mathit{Elect}(\mathit{Set}(\mathbb{N}_{nid}) * \mathit{Set}(\mathit{Log})) \\
&\quad \mid \mathit{Invoke}(\mathit{Log} * \mathit{Set}(\mathit{Log}) * \mathit{Method}) \\
&\quad \mid \mathit{Commit}(\mathit{Log}) \\
\mathit{Op}_{\text{net}} &\triangleq \texttt{elect} : \mathbb{N}_{nid} \to \Sigma_{\text{net}} \to \Sigma_{\text{net}} \\
&\quad \mid \texttt{invoke} : \mathbb{N}_{nid} \to \mathit{Method} \to \Sigma_{\text{net}} \to \Sigma_{\text{net}} \\
&\quad \mid \texttt{commit} : \mathbb{N}_{nid} \to \Sigma_{\text{net}} \to \Sigma_{\text{net}} \\
&\quad \mid \texttt{timeout} : \mathit{Set}(\mathbb{N}_{nid}) \to \mathbb{N}_{time} \to \Sigma_{\text{net}} \to \Sigma_{\text{net}} \\
&\quad \mid \texttt{deliver} : \mathit{Msg} \to \Sigma_{\text{net}} \to \Sigma_{\text{net}}
\end{aligned}
$$

Figure 6.16: Abstract network-based state and operations.

Each replica maintains a local timestamp, a log of methods tagged with a timestamp and set of voters, and a phase, which influences what actions they are allowed to take. For example, only a replica in the *Elected* phase can `invoke` a method. Messages are divided into three variants that each contain a sender, potentially multiple recipients, a timestamp, and a command. A *Request* represents typical one-to-one communication, but as it is common for a leader to broadcast requests to every replica, it is convenient to use *RequestMany* to treat a request as if it arrives at each of its recipients simultaneously. Acknowledgements follow the dual pattern where a leader gathers responses from multiple replicas, which we model as *Ack* having multiple senders.

Requests are generated by `elect`, `invoke`, `commit`, and `timeout`. These are simple state transition functions that update the caller's local state, advance to the next phase, and

```
send(st, msg) ≜
  let sent′ = sent(network(st)) ∪ {msg} in setNetwork((sent′, recvd(network(st))))
broadcast(st, msg) ≜ send(st, msg)
commit(nid, st) ≜
  let req = RequestMany(nid, allReplicas, time(st, nid), Commit(log(st, nid))) in
  let st′ = setPhase(st, nid, WaitForAcks) in broadcast(st′, req)
doCommit(rep, t, lg) ≜ setPhase(setLog(setTime(rep, t), lg), VotedCommit)
deliver(msg, st) ≜
  if msg = RequestMany(from, to, t, Commit(lg)) then
    let recips = {doCommit(rep, t, lg) | rep ∈ to ∧ canCommit(rep, t, lg)} in
    let ack = Ack(recips, from, t, Commit(lg)) in
    send(setReplicas(st, recips), ack)
  else ...
```

Figure 6.17: The `commit` network state transition function and request handler.

create and send a request. For example, `commit` (Figure 6.17) advances from the *Invoked* phase to *WaitForAcks* and broadcasts a *RequestMany* containing the log to commit.

The `deliver` operation nondeterministically decides when a message arrives at some subset of its recipients, which models the possibility that messages may be dropped or delayed. Upon delivery, a handler is triggered for each recipient depending on the type of the message (e.g., *doCommit*). For `commit`, the recipients that are allowed to commit do so by updating their time, log, and phase. An acknowledgement is then constructed from these recipients and sent back to the leader.

In addition to assuming *RequestMany* and *Ack* are received atomically, we also simplify the network model by insisting messages arrive in order of their logical timestamps, and messages are not duplicated. These are simple assumptions to discharge in lower layers by having replicas discard or wait to handle out-of-order messages, but omitting those

$$byzCommit(rep, t, lg) \triangleq \mathbb{O}_{byz}(rep, t, lg) = rep' \implies log(rep') = log(rep) \vee log(rep') = lg$$

Figure 6.18: The byzantine `commit` request handler.

implementation details from this specification reduces the number of cases to consider and simplifies the refinement.

Modeling Byzantine Behaviors The type of specification in Figure 6.17 does not work for byzantine replicas because we cannot trust them to follow the rules of the protocol. Instead, we model them with an oracle ($\mathbb{O}_{byz}$) that can do anything within certain limits. For example, if a byzantine replica receives a `commit` request, we cannot assume it will update its log or send an acknowledgement, though it is allowed to. Nevertheless, we do know that it cannot invent logs from nothing because they are cryptographically authenticated. Therefore, the specification only requires that it either takes the proposed log, or keeps its old log (Figure 6.18).

6.5.2 Refinement Proof

The refinement proof proceeds by induction on a trace of network events, showing that every action has a corresponding AdoB step that preserves the refinement relation ($\mathbb{R}$). This relation defines the correspondence between AdoB (Σ_{AdoB}) and network (Σ_{net}) state. It consists of 15 conjuncts, the most of important of which is *LogMatch*, the relation between replicas' local logs and branches of the cache tree (Figure 6.19). *LogMatch* states that every local log corresponds to some branch of the tree, and, for honest replicas, this cache much be at least as recent as its latest supported *CCache* (*activeC*). In other words, every honest

$$
\begin{aligned}
\mathit{branchToLog}(tr, C) &\triangleq \{(t, Q, M) \mid \mathit{MCache}(\_, t, Q, M) \in \mathit{ancestors}(tr, C)\} \\
\mathit{LogMatch}(st_{net}, st_{AdoB}) &\triangleq \forall nid.\, \exists C.\, \mathit{log}(st_{net}, nid) = \mathit{branchToLog}(\mathit{tree}(st_{AdoB}), C) \\
&\quad \wedge\ nid \in \mathit{honest} \implies C \geq \mathit{activeC}(\mathit{tree}(st_{AdoB}), nid) \\
\mathbb{R}(st_{net}, st_{AdoB}) &\triangleq \mathit{LogMatch}(st_{net}, st_{AdoB}) \wedge \ldots
\end{aligned}
$$

Figure 6.19: The *LogMatch* component of the refinement relation.

replica's local log contains the same prefix of committed methods as its corresponding cache tree branch.

Note that $\mathbb{R}$ does not require a replica's local log to exactly match its active (most recently supported) cache. This is necessary because the two can temporarily drift out of sync due to incomplete operations at the network level that are modeled as no-ops in AdoB (e.g., a failed commit operation that is delivered to only one replica). In this sense, AdoB's cache tree is an under-approximation of the network-level local logs; however, *LogMatch* ensures that the two abstractions agree at least on the committed methods, which is sufficient for safety and liveness.

Combined with AdoB's safety property that *CCaches* all lie on the same branch, $\mathbb{R}$ implies that at least a quorum of honest replicas always agree on the same sequence of committed methods. Similarly, as long as GenJolteon follows a productive strategy (Definition 7) and a fair rotating leadership (Definition 9), its liveness follows from AdoB's because some replica's *activeC* is guaranteed to increase within some finite time. Furthermore, because GenJolteon uses the same *isQuorum*, *isSQuorum*, and *isMQuorum* parameters as AdoB, it inherits the ability to be instantiated to either a benign or byzantine setting.

The remaining 14 conjuncts are supporting invariants that are used to show that *LogMatch* is preserved at each step. For example, *CommitDelivered* ensures that every

CCache has a corresponding delivered `commit` request. Together with the uniqueness of `commit` requests, this can be used to show that when a `commit` request is delivered, there is not already a *CCache* for it. See Appendix B.3 for an explanation of each conjunct.

Proof Effort The refinement proof for the simplified network-based model took approximately two person-months, 900 lines of specification, and 4000 lines of proof. Much of this time was spent discovering and adjusting the 14 supporting invariants of $\mathbb{R}$. This also includes the time to adjust AdoB and its safety and liveness proofs as the refinement revealed some subtle inconsistencies between GenJolteon and AdoB's handling of timeouts (see Section 6.6.1 for details).

6.5.3 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 shim multiplexes, filters, and collects network messages so that they match the expectations of the network-level specification. For example, *RequestMany* is multi-cast to all replicas, and individual acknowledgements are coalesced into an *Ack*. The shim is also responsible for signing and validating threshold-signature-based quorum certificates so that the malicious behaviors of byzantine replicas can be detected and contained.

We evaluated the extracted code on a local cluster with a four-replica configuration. It is not optimized for performance but exhibits an average latency of 2.34 ms (excluding

cryptographic signing) to commit a request under a steady state. This is comparable to the latency 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]. 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. For liveness, we must also assume that honest replicas' clock speeds are within a reasonable bound so that after GST an honest leader has time to commit before timing out.

6.6 Discussion

6.6.1 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 it was not until we attempted to prove refinement with GenJolteon that we discovered subtle mistakes related to the handling of timeouts (Figure 6.20).

One bug came from incorrectly conflating voters and supporters of a timeout. Recall that a timeout is successful when some replica receives a super quorum of timeout messages containing each timed-out replica's current log. These messages are bundled together to form a *TC*, which acts as evidence that it is safe to begin a new round using the latest of

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

ValidOracleTimeout

$$\frac{isSQuorum(\boxed{Q}) \qquad canTimeout(tree(st), C_{max}, \boxed{Q}) \\ \forall s \in \boxed{Q} \cap honest.\, times(st)[s] \leq t \qquad \exists s \in \boxed{Q} \cap honest.\, times(st)[s] = t}{\mathbb{O}_{op}(st, nid) = Timeout(\boxed{Q}, C_{max}, t)}$$

Figure 6.20: An incorrect early attempt at modeling timeouts. The mistakes (marked with a blue box) are using one set of replicas (Q) for both the voters and supporters instead of separate Q_{vote} and Q_{supp}, and requiring C_{max} to be an *activeC* in *canTimeout*.

the contained logs. In AdoB, the TC is represented by a $TCache$, and an oracle determines what super quorum of replicas timed out.

This super quorum becomes the $TCache$'s voters, but initially they were also defined to be its supporters. However, this implies that the replicas that time out are exactly the same replicas that receive the completed TC, and this is not always the case. Indeed, the two sets can be entirely disjoint. 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 problem is solved by returning two sets from the oracle: a set of voters (Q_{vote}) that represents the replicas that timed out and a set of supporters (Q_{supp}) that observe the completed $TCache$. Q_{vote} must be a super quorum, but Q_{supp} can be as small as a single honest replica. The reason Q_{supp} must include at least one honest replica is that, otherwise, liveness is not guaranteed. Suppose again that S_1 and S_2 time out and only S_3 receives their messages and forms a TC. If S_3 is byzantine, it may decide not to send the TC to the next leader, causing the current round to never end. Fortunately, the partial synchrony

assumption prevents this case by guaranteeing that eventually some honest replica will also receive the timeout messages and form a *TC*.

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' *activeC*, but that it was also equal to one of these *activeC*. 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 Figure 6.7). 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.

These wrong paths illustrate that, although generalized consensus seems very natural in AdoB, arriving at the correct abstraction is far from trivial. As with many bugs, these were simple to fix, but identifying them in the first place was only possible with the scrutiny required by refinement.

6.6.2 AdoB Generality

We have demonstrated that AdoB is generic in the sense that it describes both benign and byzantine consensus. By not fixing the quorum and super quorum sizes, it also supports a variety of strategies, including the typical 1/2 and 2/3 majorities, as well as proof-of-stake-

style weighted majorities. A third dimension to AdoB's generality is in the protocols that implement it.

Although we only discuss refinement with GenJolteon, the same is possible for other benign and byzantine consensus protocols such as Paxos, Raft, PBFT, or Tendermint [Buchman 2016]. At their core, these protocols have the same election, pre-commit (implicit for Paxos and Raft), and commit phases and rely on overlapping quorums to guarantee agreement. The differences are all in how they represent and communicate state.

For instance, in HotStuff and Jolteon, the leader is responsible for convincing the replicas that a command is safe to commit by constructing a *QC*. In Tendermint, on the other hand, replicas gather their own evidence by broadcasting their votes to everyone instead of just the leader. There are performance implications to these communication patterns, but the result is the same from AdoB's perspective: a replica only commits a command for which it has observed a super quorum of votes.

Recall from Section 2.1.4 that certain byzantine protocols implement a pipelining optimization in which the pre-commit and commit phases are merged. Supporting this poses a slight problem for AdoB because it assumes separate `invoke` and `push` operations. One solution is to create a new pipelined 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, which would allow it to enjoy safety and liveness guarantees without having to re-prove them.

6.7 Summary

Despite different failure assumptions and communication patterns, benign and byzantine consensus bear a strong intuitive resemblance. AdoB formalizes this intuition and shows that they are, in fact, similar enough to be described by a single instance of the ADO model, thereby distilling consensus to its core elements. With this unified abstraction, we were able to complete the first mechanized safety and liveness proofs for both failure models simultaneously. The liveness proof in particular uncovered many subtle aspects of reasoning about temporal properties in a partially synchronous system. One of the key strengths of AdoB, and the ADO model in general, is that it highlights these challenges and their solutions by cleanly separating them from implementation details that would otherwise obscure them.

Chapter 7

Related Work

7.1 Abstract Models

Concurrent Memory/Object Models The ADO model is heavily influenced by prior work on shared-memory concurrent objects such as CCAL [Gu et al. 2016, 2018] and the push/pull memory model. Many distributed protocols naturally split into three phases, which map onto `pull` (get the current state and permission to change it), `invoke` (update the state), and `push` (commit the changes).

There are also parallels between the ADO model and both distributed and shared-memory transactional models [Guerraoui and Kapalka 2008; Koskinen and Parkinson 2015]. A successful `push` behaves similarly to a transaction commit in that it atomically commits (a prefix of) the active cache tree branch while simultaneously aborting inconsistent states in sibling branches. However, transactions typically rely on a centralized coordinator to ensure that updates are applied to the latest consistent snapshot, while the ADO model is more decentralized and allows `pull` to select an inconsistent state as a starting point. The

Replicated State Safety property guarantees that these inconsistent states are descendants of the latest committed state, but there may temporarily be "competing" snapshots until they are resolved by push.

Distributed Object Models Wang et al. [2019] showed that conflict-free replicated data types (CRDTs) [Shapiro et al. 2011] that satisfy a property called replication-aware linearizability can be modeled by a modular, sequential specification. This is similar to the ADO model in that it hides distributed behaviors behind a compositional, atomic interface, but CRDTs offer eventual consistency whereas the ADO model targets strong consistency.

Another framework for modular reasoning about distributed systems is ModP [Desai et al. 2018]. In this language, systems are modeled as concurrent state machines encapsulated within a module. Communication is modeled by sending events to other modules, which can trigger state transitions, similar to method calls. A module can refer to an abstract interface, which can be instantiated by composing with another module that implements the interface. Unlike the ADO, this model is not designed for arbitrary high-level reasoning, but rather for improving the scalability of distributed system testing by allowing modules to be tested in isolation.

Another common object-like abstraction for distributed systems is state machine replication (SMR) with remote procedure calls (RPC), which hide intermediate states due to transient failures, often by wrapping methods in a retry loop [Schneider 1990]. This can be convenient, but it prevents reasoning about applications with alternate failure-handling strategies (e.g., at-most-once calls), those that use inconsistent states (e.g., TAPIR [Zhang et al. 2015]), or those with optimizations that do not follow the typical message-sending

	Benign	Byz.	Safe	Live	Exec.	Refine.
AdoB	✓	✓	✓	✓	✓	✓
Advert	✓	×	✓	×	✓	✓
Adore	✓	×	✓	×	✓	✓
IronFleet [Hawblitzel et al. 2015a]	✓	×	✓	✓	✓	✓
Verdi [Wilcox et al. 2015]	✓	×	✓	×	✓	✓
Velisarios [Rahli et al. 2018]	×	✓	✓	×	✓	×
Carr et al. [2022]	×	✓	✓	×	×	×
Losa and Dodds [2020]	×	✓	✓	✓	×	×

Table 7.1: Comparison of selected consensus verification projects.

patterns (e.g., 2PC with consensus [Gray and Lamport 2006]). The ADO model does support these types of applications, but one can also easily recover SMR-like behaviors by using exactly-once calls when this level of control is unnecessary. This means one can build an application that mixes SMR-style objects with those that exploit intermediate states, and reason about their interactions using the common ADO foundation.

7.2 Formal Verification

7.2.1 Consensus

Consensus verification is a well-studied topic, with many projects of varying scope. Table 7.1 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. The first three lines answer these questions for the case studies presented in this dissertation and show that AdoB is the only one to cover all of the points and is also the only one to support benign and byzantine consensus. Nevertheless,

ADVERT and ADORE each have other advantages that are not considered in this table, such as application-level reasoning and supporting reconfiguration.

IronFleet IronFleet [Hawblitzel et al. 2015a] is a verification framework implemented in Dafny [Leino 2010]. It adopts a layered verification approach in which one begins by writing a network-level specification and annotating it with Hoare-logic-style pre and post-conditions, which can be automatically validated by an SMT solver. Then, one writes another slightly more abstract specification and uses a series of reduction arguments to reorder, remove, or join sequences of network events into simpler ones, much like the network refinements presented in Sections 4.5, 5.5, and 6.5.

After the refinements, one can prove high-level properties about an application in something more akin to an SMR model. The use of SMT solvers to automatically discover proofs can be very useful; however, it is more limited when considering higher-order properties involving quantifiers. These cases often require clever developer-provided heuristics to guide the proof search.

Of the selected benign verification frameworks, IronFleet is the only one to support liveness reasoning. This is enabled through an embedding of the temporal logic of actions (TLA [Lamport 1994]), which allows one to state that a property holds *eventually* or *always*. Unlike ADOB, IronFleet's liveness proofs are application-specific (e.g., a particular service will eventually respond to a client request), rather than generic properties that hold for an entire class of protocols.

Dafny also supports unverified compilation to C#, and the related Ironclad [Hawblitzel et al. 2014] work demonstrated that Dafny specifications can be translated into BoogieX86

verifiable assembly [Barnett et al. 2005] as well. Unlike the ADO model, IronFleet's strengths lie more in facilitating application-specific reasoning rather than providing a generic, reusable protocol-level abstraction.

Verdi Verdi [Wilcox et al. 2015] is a Coq distributed system verification framework in which one writes an application in a network-based style, and reasons about the traces of external events it generates. Like IronFleet, Verdi supports a layered style where complex implementations are refined into simpler specifications, which can then be used as a basis on which to build additional layers.

To help with this process it provides a set of verified system transformers (VSTs), which can automatically and safely transform one specification into another. This allows one, for example, to write a naïve system that assumes a reliable network, and automatically generate a more robust implementation with fault-tolerance mechanisms, such as sequence numbers to deduplicate messages. The VST also generates a refinement proof that transfers properties about the simpler system to the more realistic one.

Verdi does not support temporal reasoning. Because it is implemented in Coq, it allows specifications to be extracted to OCaml. As with IronFleet, Verdi does not provide a common atomic abstraction for consensus like the ADO model, but instead provides developers with tools to reason about individual systems in a more ad-hoc manner.

Velisarios 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. In some ways, the ADO model is a hybrid between this abstraction and a network-based model in that it also captures the history of a distributed system as a collection of events with dependencies; however, the tree-based structure makes the relation to the concrete state (i.e., logs of commands) more explicit.

Unlike AdoB, Velisarios does not consider benign consensus or liveness. Additionally, although it provides a Coq specification of PBFT that can be extracted to OCaml, it is not proved to refine the logic-of-events model. As Section 6.6.1 shows, this specification gap can leave the possibility of bugs even in otherwise verified systems.

HotStuff Carr et al. [2022] proved the safety of a generalized abstract specification of HotStuff in Agda [Agda Development Team 2005–2022]. 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 the ADO model's goal of capturing the core behaviors of a protocol so proofs of high-level properties can be reused for many implementations; however, this work's scope is much narrower as it is targeted specifically at HotStuff variants and does not cover benign consensus, nor does it prove liveness or provide a verified executable.

Stellar Losa and Dodds [2020] are the first to mechanically prove both the safety and liveness of a byzantine protocol, Stellar. Instead of traditional (super) quorums, Stellar uses a slightly different technique to reach consensus called federated agreement in which each replica makes an individual decision about which replicas it trusts (called a quorum

slice). The proof uses a Coq framework 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 1986–2022] by showing that axioms in the Coq model hold in Isabelle. However, there is no mechanically-checked connection between the models nor is there any connection to an executable implementation. This work is also specific to Stellar and does not prove anything about benign or byzantine consensus in general.

7.2.2 Proof Automation

There is also a large body of work on techniques for transforming an asynchronous program into into an equivalent sequential one by logically rearranging traces of communication operations using mover types [Chajed et al. 2018; Hawblitzel et al. 2015b; Kragl et al. 2020; v. Gleissenthall et al. 2019]. Some of these tools can create a sequentialized program automatically, while others provide a library of validated transformations that can be applied manually. This can be very powerful for reducing verification complexity, but they often only support specific communication patterns, and some do not handle server and network failures.

Other work on proof automation includes Ivy [Padon et al. 2016], a verification toolkit that combines interactive and automation techniques to prove safety properties of distributed protocols modeled as unbounded state machines. Padon et al. [2017] and Taube et al. [2018] build on Ivy to introduce methodologies for automatically proving properties about systems that satisfy certain decidability conditions. I4 [Ma et al. 2019] explores

an incremental process for automatic generation of distributed system invariants. These projects aim to ease network-based reasoning and are largely orthogonal to the ADO model's goal of providing a general protocol-level abstraction; however, they may be useful for simplifying or automating network-level refinement proofs.

7.2.3 Composition

One shortcoming of most distributed system verification frameworks, including Verdi and IronFleet, is a lack of support for composition between applications and clients, which limits the modularity and reuse of verified systems. The ADO model supports this with DApps (Section 4.4), which define how clients can interact with a set of ADOs.

Disel [Sergey et al. 2017] is another framework that supports decomposing composite applications into isolated pieces. Systems are defined with a shallowly-embedded language in Coq, which makes use of dependent types to allow users to define protocol invariants that must always hold. Users then prove these invariants using a program logic built on a distributed variation of concurrent separation logic.

Interaction between components is modeled by *send-hooks*, which allow a user to limit allowed communication and define conditions that must be satisfied. Compared to the ADO model, the component specifications are at a much lower level of abstraction that exposes some protocol implementation details. This ties the reasoning to explicit network communication patterns rather than the generic ADO `pull-invoke-push` interface.

Another line of work on distributed system composition using separation logic is Aneris [Krogh-Jespersen et al. 2020]. It provides a higher-order ML-like specification

language and the program logic is built on top of the Iris verification framework [Jung et al. 2015, 2018]. Notably, it supports thread-level concurrency within a single replica in addition to the usual inter-replica concurrency. Like Disel, it defines interactions at the level of abstract network primitives and cannot support the kind of network-independent reasoning enabled by the ADO model.

Both frameworks can handle composition examples such as Two-Phase Commit, which rely on a coordinator to manage concurrency; however, to our knowledge they have not demonstrated support for more decentralized applications such as `KVLockFree`.

7.3 Partial Failures

One advantage of the ADO model is its simple interface for working with one of the most complicated and unintuitive aspects of distributed systems: failures and intermediate state. As our replicated Two-Phase Commit example (Section 4.4.2) demonstrates, one can exploit these features for performance gains. For example, by using `pull` and `push` directly instead of an exactly-once call, the TM is able to save several message round trips.

TAPIR [Zhang et al. 2015] also combines transactions with consensus, but it observes that, since both the transaction and replication protocols are strongly consistent, it can replicate commands with only a single round of messages instead of two. This means replicas may receive commands in different orders, but the consistent global order is enforced later by the TM. We could model this behavior with cache tree entries for the replicated commands, which are only committed later by `push`. Much like the Two-Phase Commit example, by carefully controlling when `pull` and `push` are called and temporarily

relying on uncommitted states, TAPIR exploits an application-specific characteristic (the existence of the TM) to optimize its performance.

Another use for exposed failures is speculative execution. Speculator [Nightingale et al. 2006] is a distributed file system that outperforms NFS by working under the assumption that its operations will succeed without waiting for confirmation. If it later learns that an operation failed, it reloads from an earlier checkpoint and retries. The speculation and failures are hidden from the client by waiting to externalize the output until an operation succeeds, but, in order to make this optimization possible, they must be exposed within the boundaries of the application.

At present, these systems are quite complicated and not well supported by existing models. Thus, in spite of the potential performance benefits, it is difficult for developers to be confident in the correctness of their implementations. The ADO model may make formal verification of such systems much more feasible, which could encourage their development and adoption.

7.4 Reconfiguration

7.4.1 Alternate Reconfiguration Schemes

Adore is designed to support hot reconfiguration algorithms where uncommitted reconfiguration commands update the configuration immediately. These are both more efficient and more challenging to verify than other types because they interleave reconfiguration with normal operations. The cache tree's representation of uncommitted state is ideal for

handling this kind of speculative behavior, and the $R1^+$ and *isQuorum* parameters allow Adore to support a wide variety of schemes; however, with some slight modifications, it could easily handle even more.

Lamport et al. [2008] suggests an "easy" approach to reconfiguration in which each instance of consensus (each slot in the log) uses a configuration that is inherited from the previous instance. The configuration can then be changed by committing a special command that tells the next consensus instance to use it. To avoid blocking instance $i + 1$ from beginning until i is fully committed, the algorithm is generalized to use a parameter α such that a configuration committed in instance i takes effect in instance $i + \alpha$, thus allowing i through $i + \alpha - 1$ to continue as normal.

This scheme already has a fair amount in common with Adore, such as inheriting the previous slot's configuration. In order to fully support it, the first required change is to wait until a configuration is committed to begin using it, rather than having it take effect immediately. The other is to block new methods from being invoked on an active branch that has α uncommitted caches.

Stoppable Paxos [Malkhi et al. 2008] introduces a "stop" command that prevents replicas from committing further commands. Once stopped, a new configuration is launched and the old log is copied over. WormSpace [Shin et al. 2019] implements a similar approach by "sealing" the configuration, which causes servers to reject subsequent requests, before starting a new instance. Liskov and Cowling [2012] also define a membership change algorithm for Viewstamped Replication in which a special command defines the new configuration and begins a view change (i.e., election). Like the other approaches, handling of client requests is paused until the logs are completely transferred to the new configuration.

Adore could model this style of stop-the-world reconfiguration by deleting all caches not on the active branch when an *RCache* is committed, which simulates copying the committed commands to a new cluster of servers. This simplifies the problem because once the *RCache* is committed there is a clean break between the old and new configurations with no opportunity for both to run simultaneously.

7.4.2 Formal Verification

There is surprisingly little prior work on formal verification of reconfiguration. Verdi's proof of Raft's safety [Woos et al. 2016] does not consider either the single-server or joint consensus algorithms. IronFleet's Paxos-based IronRSL also omits reconfiguration though they claim it "only requires additional developer time". Padon et al. [2017] prove the safety of Vertical Paxos but assume the existence of a correct external reconfiguration service, thus sidestepping the issue. Even in the blockchain world, where membership tends to be very flexible, verification efforts often make strict assumptions about configurations. For example, Losa and Dodds [2020] prove the safety of the Stellar Consensus Protocol, but assume an "arbitrary, but fixed configuration". Likewise, Carr et al. [2022] only consider a single epoch of HotStuff in which the set of validators does not change.

The work nearest to Adore is the verification of the reconfiguration scheme used by MongoDB [Schultz et al. 2022a,b], which occurred concurrently with Adore's development. Like Adore, the protocol is based on Raft's single-server algorithm, but with an optimization that reconfiguration operations are stored separately from regular commands, which somewhat relaxes the dependencies between them, and means replicas only need

to keep the latest configuration. However, there are several significant differences in our approaches and results.

The MongoDB work is specified in TLA$^+$ [Lamport 1999, 2002] in a very abstract network-based model and its safety is proved with the TLA$^+$ proof system (TLAPS) [Chaudhuri et al. 2008]. The specification is at a comparable level of abstraction to our SRaft specification (Section 5.5), where communication details are mostly hidden but state is still modeled as local logs rather than a global cache tree. Unlike Adore, there is no refinement with a more realistic model or the ability to extract executable code. The MongoDB specification is also for a fixed reconfiguration scheme and lacks the generality of Adore's *isQuorum* and R1$^+$ parameters.

The high-level structures of the safety proofs are quite different as well. Because MongoDB also uses a hot reconfiguration algorithm, it faces the same sort of circularity problems described in Section 5.4; however, without Adore's tree structure to suggest the *rdist*-based approach, they resort to the more standard technique of establishing an inductive invariant that implies safety and is preserved by every step of the specification. This invariant must contain enough information both to prove safety and its own invariance, which means several mutually dependent properties must be bundled together. In particular, the MongoDB invariant is a conjunction of 20 high-level properties ranging from important safety guarantees, like election safety and log matching, to implementation details, such as the uniqueness of a configuration's term and version number.

With all of these invariants packed together, the intuition behind why the protocol works is obscured, and the proof is more complex because it is harder to break down into smaller steps. Discovering the correct invariants alone took between one to two person-

months using a counterexample-driven approach with a tool that attempts to detect invalid invariants. Actually proving that the invariant is inductive and implies safety took another four person-months. When compared with the five person-weeks to prove Adore's safety, this supports our claim that finding the correct protocol-level abstraction is essential for scaling verification to more realistic and complex systems.

7.5 Connecting Benign and Byzantine Consensus

Although AdoB is the first fully mechanized abstraction to prove the safety and liveness of benign and byzantine consensus simultaneously, others have also noted the similarities between the failure models and attempted to formalize the connection. Lamport [2011] demonstrated that a byzantine version of Paxos (BPCon) refines a modified version of benign Paxos (PCon). In particular, PCon adds a 1*c* message (pre-commit in our terminology) that asserts a particular value is safe to commit. In PCon, this message is purely logical because it is implied by the following 2*a* (commit) message and the leader is trusted, but in BPCon, it is essential to ensure a byzantine leader cannot convince honest replicas to commit an invalid state. PCon is proved to be safe in TLAPS and is then "byzantinized" by proving that BPCon refines it, showing that both implement consensus despite the presence of malicious replicas.

The 1*c* 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. Lamport provides an informal argument for the liveness of BPCon, but no mechanized proof.

In a similar vein, some of the HotStuff designers have informally described how to derive a benign version of HotStuff by dropping the pre-commit phase and cryptographic signatures, and using a smaller quorum [Abraham et al. 2021]. The fact that such a simple transformation is possible is part of what inspired the use of elements of HotStuff's design (e.g., rotating leadership every round, quorum and timeout certificates) in AdoB.

Another more general approach is proposed by Rütti et al. [2010], which aims to provide a generic specification for benign and byzantine consensus. Once again, the key to this unification is to parameterize the pre-commit phase (what they refer to as the validation round) to require more or less evidence from the leader that a command is safe to commit. In particular, the decision (commit) round is controlled by a parameterized threshold of required votes (T_D), and *FLAG*, which decides if the votes must be validated or not.

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, the authors do not provide mechanized proofs of safety or liveness for the generalized consensus algorithm. Furthermore, it is specified in terms of a very abstract network-based model with no formal connection to an implementation.

Chapter 8

Conclusions and Future Work

The goal of this dissertation has been to present the atomic distributed object abstraction and demonstrate its effectiveness for simplifying many aspects of distributed system verification. This is shown through a series of case studies, each of which develops a variant of the ADO model and uses it to solve a verification challenge.

First, Advert explores using ADOs for application-level reasoning and composing them into larger systems. Next, Adore shifts the focus to proving the safety of consensus protocols with the added complexity of a generic hot reconfiguration scheme. Finally, AdoB introduces liveness reasoning and generalizes the scope from benign consensus to a unified model that covers both benign and byzantine failures.

Each case also demonstrates the validity and generality of the ADO model by proving refinements with a variety of protocols including several Paxos variants, Chain Replication, a version of Raft, and a version of Jolteon. These illustrate that the ADO model, although primarily designed for high-level reasoning, is also a viable tool for producing systems that are both practical and verified.

This is a promising line of research with many possible future directions, some of which are outlined below.

8.1 Combining ADO Variants

We have shown several variants of the ADO model that target different verification challenges. It would be worthwhile to combine or otherwise link these into a more cohesive abstraction that covers all cases. For Adore and AdoB, this would mean defining a model with *RCaches*, *TCaches*, and the generalized benign/byzantine failure model. We expect that much of the safety and liveness proofs would be unaffected, but the interaction between reconfiguration, timeouts, and byzantine replicas will likely introduce new complexities.

For Advert, on the other hand, it is not clear that merging it with Adore or AdoB would be desirable because its focus is on a slightly higher-level of abstraction. Instead, it would be better to prove that Adore and AdoB refine Advert, which would allow Advert's DApps to benefit from the lower-level models' safety and liveness proofs without complicating them with unnecessary details. This proof should be relatively straightforward as Advert's cache tree is very similar to that of Adore and AdoB but with less information about voters. The primary challenge would be mapping this information instead to the *ActiveMap*, which is a more coarse-grained method for tracking active caches.

8.2 Alternate Consistency Models

In this work we have focused on consensus, which provides strong consistency for the replicated state; i.e., all operations are guaranteed to be observed in the same order. This is a very intuitive and often necessary behavior, but it often comes at the expense of high availability [Gilbert and Lynch 2002]. In practice, applications may choose to use consensus to maintain only the most critical data, and a weaker model that trades consistency for improved responsiveness everywhere else [Chang et al. 2006; Dean 2009].

For example, the eventual consistency model, which is used by conflict-free replicated data types (CRDTs), may temporarily allow different clients to see different views of the replicated state but guarantees that eventually they will converge. To support a wider range of distributed systems, the ADO model could be modified to handle this or other consistency models. This would require weakening the effect that *CCaches* have on `pull` because it is no longer guaranteed that exactly one branch is committed at all times. However, if enough of the abstraction remains the same, it could perhaps even support composing objects with different consistency models, which would enable verification of the kind of heterogeneous applications that appear in practice.

8.3 Proof Automation

Network Refinement Despite simplifying reasoning about distributed protocols compared to network-based models, proofs in the ADO model still require a significant amount of manual effort. It would be interesting to explore what aspects of these proofs could be fully or partially automated by integrating with existing verification tools. One particularly

promising area to apply this would be the network refinement proofs. The process of reordering and grouping asynchronous network events into a canonical form is precisely what Verdi's VSTs are designed for. One could, in principle, use Verdi to prove that a realistic network-based model refines a much simpler atomic model, which would only leave the final step of connecting the atomic network model to the ADO model.

Even with proof automation tools, it can still be challenging to simply write a correct specification. In fact, much of the time to complete the network refinement proofs was spent writing a specification, attempting the proof only to find it is impossible, adjusting the specification, and repeating until the process converges. Instead of doing these steps entirely manually, this cycle could be accelerated with property-based testing. For example, a tool like QuickChick [Lampropoulos and Pierce 2018] can quickly generate a large number of random test inputs (e.g., traces of network events) and automatically check if they satisfy a set of desired properties (e.g., the refinement relation). Any counter-examples can be studied to discover specification errors, which is much simpler than attempting to identify the problem from deep within a proof goal.

Executable Code Generation Another area for improvement is the generation of verified executable code. For Advert, we manually proved a refinement with hand-written C code using CCAL, which demonstrates that end-to-end verification of efficient distributed programs is possible, albeit very time consuming. A much simpler alternative is the approach taken in Adore and AdoB of automatically generating OCaml code directly from the Coq specification. Unfortunately, this introduces a much larger trusted computing base and may not satisfy the performance requirements of certain applications.

A promising solution that combines the best of both options is the DeepSEA language [Sjöberg et al. 2019], which automatically produces C code from a high-level functional specification along with a refinement proof. The refinement proof may, in some cases, leave a few verification conditions to be proved manually, but this would still represent a significant reduction in proof effort. Combined with Verdi or a comparable alternative for the network refinement, this would make it much easier to produce even more optimized and realistic verified distributed systems.

Model Checking A third option for introducing automation is to use ADO model as a basis for model checking. Model checking is a common testing technique for distributed systems that takes an abstract representation of a system, typically as a very high-level network-based model or state machine, and a property to prove, and automatically searches for counterexamples [Lamport 1999]. Except for very simple or very regular systems, exhaustive search is impossible, so it cannot guarantee the absence of a bug; however, it can still be a useful sanity check that is much lower-effort than full-scale verification. It would be interesting to apply these techniques to a system that is specified in terms of the ADO model. Because the interface consists of only three operations with relatively few potential outcomes, it may be possible to explore a larger search space than with a network-based model.

8.4 Addressing Implementation Inefficiencies

Although we have shown that the ADO model can be refined by several consensus protocols, the examples have been optimized for ease of verification rather than performance. This is not a fundamental limitation of the ADO model, but implementing more efficient systems complicates the refinement proofs. For example, in our Raft and Jolteon implementations, replicas send their entire log in each request. In practice, this becomes prohibitively inefficient as the length of the log grows. Instead, systems typically implement techniques to reduce message sizes, such as sending only the log entries that a replica is missing or sending just a hash of the log and relying on an external protocol like Narwhal [Danezis et al. 2022] for data replication.

This type of optimization is invisible at the ADO level, as replicas are simply assumed to have access to the methods in their active branch. A benefit of this design is that data replication concerns are isolated from safety and liveness properties, but the downside is it is less clear how these operations map onto the ADO behaviors. A brute-force solution is to simply add more layers of network refinement, and perhaps, with the help of automation techniques, the proofs can be made feasible.

An alternative is to investigate whether an intermediate model between the network and ADO levels might be able to bridge the gap and reduce the proof burden. For instance, something similar to the ADO model could be equipped with a `replicate` command that must be called before a replica switches its active branch. Because this intermediate model also has a cache tree-based design, the refinement with the ADO model would be simplified, and the new operation makes it easier to define the relation with a network-

based specification as well.

This is similar to how Adore and AdoB can be seen as lower-level versions of Advert with additional information about the configuration. By continuing to design alternate ADO variants with increasingly low-level system details, one can break the relation between the highest and lowest specifications into more manageable steps, while still preserving the isolation between abstraction layers that makes the ADO model so effective at protocol-level reasoning.

Bibliography

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

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

Andrew W. Appel. Verified software toolchain. In Gilles Barthe, editor, *Proceedings of the European Symposium on Programming*, ESOP '11, Berlin, Germany, March 2011. Springer. doi: https://doi.org/10.1007/978-3-642-19718-5_1.

AWS Team. Summary of the Amazon EC2 and Amazon RDS service disruption in the US east region, April 2011. URL https://aws.amazon.com/message/65648/.

Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: A shared log design for flash clusters. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, NSDI '12, Berkeley, CA, USA, April 2012. USENIX Association. doi: https://doi.org/10.1145/2535930.

Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Proceedings of the International Symposium on Formal Methods for Components and Objects*, FMCO '05, pages 364–387, Berlin, Germany, November 2005. Springer. doi: https://doi.org/10.1007/11804192_17.

Christian Berger, Hans P. Reiser, and Alysson Bessani. Making reads in BFT state machine replication fast, linearizable, and live. In *Proceedings of the International Symposium on Reliabile Distributed Systems*, SRDS '21, Washington, DC, USA, September 2021. IEEE Computer Society. doi: https://doi.org/10.1109/SRDS53918.2021.00010.

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

Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making Byzantine consensus live. In *Proceedings of the International Conference on Distributed Computing*, DISC '20, Dagstuhl, Germany, July 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi: https://doi.org/10.4230/LIPIcs.DISC.2020.23.

Ethan Buchman. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*. PhD thesis, University of Guelph, 2016.

Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, November 2006. USENIX Association. URL https://dl.acm.org/doi/10.5555/1298455.1298487.

Christian Cachin and Marko Vukolic. Blockchain consensus protocols in the wild. *arXiv*, July 2017. doi: https://doi.org/10.48550/arXiv.1707.01873.

Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer, Berlin, Germany, 2 edition, 2011. doi: https://doi.org/10.1007/978-3-642-15260-3.

Harold Carr, Christa Jenkins, Mark Moir, Victor Cacciari Miraldo, and Lisandra Silva. Towards formal verification of HotStuff-based Byzantine fault tolerant consensus in Agda. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods*, NFM '22, pages 616–635, Berlin, Germany, May 2022. Springer. doi: https://doi.org/10.1007/978-3-031-06773-0_33.

Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, February 1999. USENIX Association. URL https://dl.acm.org/doi/10.5555/296806.296824.

Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Communications of the ACM*, 62(12):44–54, December 2019. doi: https://doi.org/10.1145/3368454.

Tej Chajed, Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. Verifying concurrent software using movers in CSPEC. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 306–322, Berkeley, CA, USA, October 2018. USENIX Association. URL https://dl.acm.org/doi/10.5555/3291168.3291191.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 205–218, New York, NY, USA, November 2006. Association for Computing Machinery. doi: https://doi.org/10.1145/1365815.1365816.

Kaustuv C Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. A TLA+ proof system. In Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, Renate Schmidt, and Stephan Schulz, editors, *Proceedings of the Workshop on Knowledge Exchange: Automated*

Provers and Proof Assistants, KEAPPA '08, pages 17–37, online, November 2008. CEUR-WS.org. URL http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-418/paper2.pdf.

Coq Development Team. The Coq proof assistant, 1999–2022. URL http://coq.inria.fr.

CorfuDB. CorfuDB, 2017. URL https://www.github.com/CorfuDB/CorfuDB.

George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and Tusk: A DAG-based mempool and efficient BFT consensus. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys '22, pages 34–50, New York, NY, USA, March 2022. Association for Computing Machinery. doi: https://doi.org/10.1145/3492321.3519594.

Jeff Dean. Designs, lessons and advice from building large distributed systems, 2009. URL https://research.cs.cornell.edu/ladis2009/talks/dean-keynote-ladis2009.pdf. Keynote from ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware.

Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. Compositional programming and testing of dynamic distributed systems. *Proceedings of the ACM on Programming Languages*, OOPSLA, November 2018. doi: https://doi.org/10.1145/3276529.

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

etcd Developers. etcd, 2013–2022. URL https://etcd.io/.

Pascal Felber, Ben Jai, Rajeev Rastogi, and Mark Smith. Using semantic knowledge of distributed objects to increase reliability and availability. In *Proceedings of the International Workshop on Object-Orient Real-Time Dependable Systems*, WORDS '01, pages 153–160, Washington, DC, USA, February 2001. IEEE Computer Society. doi: https://doi.org/10.1109/WORDS.2001.945126.

Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374−−382, April 1985. doi: https://doi.org/10.1145/3149.214121.

Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys '17, pages 328–343, New York, NY, USA, April 2017. Association for Computing Machinery. doi: https://doi.org/10.1145/3064176.3064183.

Eli Gafni and Leslie Lamport. Disk Paxos. *Distributed Computing*, 16(1), February 2003. doi: https://doi.org/10.1007/s00446-002-0070-8.

Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and Ditto: Network-adaptive efficient consensus with asynchronous fallback. In *Proceedings of the International Conference on Financial Cryptography and Data Security*, FC '22, Berlin, Germany, May 2022. Springer. URL https://fc22.ifca.ai/preproceedings/35.pdf.

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, October 2003. Association for Computing Machinery. doi: https://doi.org/10.1145/945445.945450.

Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, June 2002. doi: https://doi.org/10.1145/564585.564601.

Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. *ACM SIGCOMM Computer Communication Review*, 41(4):350–361, August 2011. doi: https://doi.org/10.1145/2043164.2018477.

Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems*, 31(1):133–160, March 2006. doi: https://doi.org/10.1145/1132863.1132867.

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *Proceedings of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 595–608, New York, NY, USA, January 2015. Association for Computing Machinery. doi: https://doi.org/10.1145/2676726.2676975.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 653–669, Berkeley, CA, USA, November 2016. USENIX Association. URL https://dl.acm.org/doi/10.5555/3026877.3026928.

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. In *Proceedings of the ACM SIGPLAN International Conference of Programming Language Design and Implementation*, PLDI '18, pages 646–661, New York, NY, USA, April 2018. Association for Computing Machinery. doi: https://doi.org/10.1145/3296979.3192381.

Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, New York, NY, USA, February 2008. Association for Computing Machinery. doi: https://doi.org/10.1145/1345206.1345233.

Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '14, New York, NY, USA, November 2014. Association for Computing Machinery. doi: https://doi.org/10.1145/2670979.2670986.

Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 165–181, Berkeley, CA, USA, October 2014. USENIX Association. URL `https://dl.acm.org/doi/10.5555/2685048.2685062`.

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '15, New York, NY, USA, October 2015a. Association for Computing Machinery. doi: https://doi.org/10.1145/2815400.2815428.

Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serar Tasiran. Automated and modular refinement reasoning for concurrent programs. In *Proceedings of the International Conference on Computer Aided Verification*, CAV '15, pages 449–465, Berlin, Germany, July 2015b. Springer. doi: https://doi.org/10.1007/978-3-319-21668-3_26.

Wolf Honoré, Jieung Kim, Ji-Yong Shin, and Zhong Shao. Much ADO about failures: A fault-aware model for compositional verification of strongly consistent distributed systems. *Proceedings of the ACM on Programming Languages*, OOPSLA, October 2021a. doi: https://doi.org/10.1145/3485474.

Wolf Honoré, Jieung Kim, Ji-Yong Shin, and Zhong Shao. Artifact for "Much ADO about failures: A fault-aware model for compositional verification of strongly consistent distributed Systems", September 2021b.

Wolf Honoré, Ji-Yong Shin, Jieung Kim, and Zhong Shao. Adore: Atomic distributed objects with certified reconfiguration. In *Proceedings of the ACM SIGPLAN International Conference of Programming Language Design and Implementation*, PLDI '22, pages 379–394, New York, NY, USA, June 2022a. Association for Computing Machinery. doi: https://doi.org/10.1145/3519939.3523444.

Wolf Honoré, Ji-Yong Shin, Jieung Kim, and Zhong Shao. Artifact for "Adore: Atomic distributed objects with certified Reconfiguration", March 2022b.

Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference*, USENIXATC '10, Berkeley, CA, USA, June 2010. USENIX Association. URL `https://dl.acm.org/doi/10.5555/1855840.1855851`.

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

Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Proceedings of the International Conference on Distributed Computing*, DISC '16, pages 313–327, Berlin, Germany, September 2016. Springer. doi: https://doi.org/10.1007/978-3-662-53426-7_23.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. *Proceedings of the ACM on Programming Languages*, Proceedings of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: 637–650, January 2015. doi: https://doi.org/10.1145/2775051.2676980.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, November 2018. doi: https://doi.org/10.1017/S0956796818000151.

Tom Killalea. The hidden dividends of microservices. *Communications of the ACM*, 59(8): 42–45, August 2016. doi: https://doi.org/10.1145/2948985.

Eric Koskinen and Matthew Parkinson. The Push/Pull model of transactions. In *Proceedings of the ACM SIGPLAN International Conference of Programming Language Design and Implementation*, PLDI '15, pages 186–195, New York, NY, USA, June 2015. Association for Computing Machinery. doi: https://doi.org/10.1145/2737924.2737995.

Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. Inductive sequentialization of asynchronous programs. In *Proceedings of the ACM SIGPLAN International Conference of Programming Language Design and Implementation*, PLDI '20, pages 227–242, New York, NY, USA, June 2020. Association for Computing Machinery. doi: https://doi.org/10.1145/3385412.3385980.

Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. Aneris: A mechanised logic for modular reasoning about distributed systems. In Peter Müller, editor, *Proceedings of the European Symposium on Programming*, ESOP '20, pages 336–365, Berlin, Germany, April 2020. Springer. doi: https://doi.org/10.1007/978-3-030-44914-8_13.

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

Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2): 133–169, May 1998. doi: https://doi.org/10.1145/279227.279229.

Leslie Lamport. Specifying concurrent systems with TLA+. *Calculational System Design*, 173:183–247, April 1999. URL https://www.microsoft.com/en-us/research/publication/specifying-concurrent-systems-tla/.

Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):34–58, December 2001. doi: https://doi.org/10.1145/568425.568433.

Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston, MA, USA, June 2002. URL https://www.microsoft.com/en-us/research/publication/specifying-systems-the-tla-language-and-tools-for-hardware-and-software-engineers/.

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

Leslie Lamport. Byzantizing Paxos by refinement. In *Proceedings of the International Conference on Distributed Computing*, DISC '11, pages 211–224, Berlin, Germany, September 2011. Springer. URL https://dl.acm.org/doi/10.5555/2075029.2075058.

Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982. doi: https://doi.org/10.1145/357172.357176.

Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. Technical Report MSR-TR-2008-193, Microsoft, February 2008. URL https://www.microsoft.com/en-us/research/publication/reconfiguring-a-state-machine/.

Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical Paxos and primary-backup replication. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '09, pages 312–313, New York, NY, USA, August 2009. Association for Computing Machinery. doi: https://doi.org/10.1145/1582716.1582783.

Leonidas Lampropoulos and Benjamin C. Pierce. *QuickChick: Property-Based Testing in Coq*. Software Foundations series, volume 4. Electronic textbook, August 2018. URL http://www.cis.upenn.edu/~bcpierce/sf.

K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Proceedings of the International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR '10, pages 348–370, Berlin, Germany, 2010. Springer. doi: https://doi.org/10.1007/978-3-642-17511-4_20.

Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43 (4):363–446, November 2009. doi: https://doi.org/10.1007/s10817-009-9155-4.

Hongjin Liang, Jan Hoffmann, Xinyu Feng, and Zhong Shao. Characterizing progress properties of concurrent objects via contextual refinements. In *Proceedings of the International Conference on Concurrency Theory*, CONCUR '13, pages 227–241, Berlin, Germany, August 2013. Springer. doi: https://doi.org/10.1007/978-3-642-40184-8_17.

Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012. URL http://hdl.handle.net/1721.1/71763.

Giuliano Losa and Mike Dodds. On the formal verification of the Stellar consensus protocol. In Bruno Bernardo and Diego Marmsoler, editors, *Proceedings of the Workshop on Formal Methods for Blockchains*, FMBC '20, Dagstuhl, Germany, July 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi: https://doi.org/10.4230/OASIcs.FMBC.2020.9.

Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: Incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '19, pages 370–384, New York, NY, USA, October 2019. Association for Computing Machinery. doi: https://doi.org/10.1145/3341301.3359651.

Dahlia Malkhi, Leslie Lamport, and Lidong Zhou. Stoppable Paxos. Technical Report MSR-TR-2008-192, Microsoft, April 2008. URL `https://www.microsoft.com/en-us/research/publication/stoppable-paxos/`.

Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. A large scale study of data center network reliability. In *Proceedings of the Internet Measurement Conference*, IMC '18, pages 393–407, New York, NY, USA, October 2018. Association for Computing Machinery. doi: https://doi.org/10.1145/3278532.3278566.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978. doi: https://doi.org/10.1016/0022-0000(78)90014-4.

Atsuki Momose and Jason Paul Cruz. Force-locking attack on sync hotstuff. *Cryptology ePrint Archive*, (2019/1484), January 2020. URL `https://eprint.iacr.org/2019/1484`.

Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, November 2013. Association for Computing Machinery. doi: https://doi.org/10.1145/2517349.2517350.

Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. *ACM Transactions on Computer Systems*, 24(4):361–392, 2006. doi: https://doi.org/10.1145/1189256.1189258.

Diego Ongaro. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, 2014.

Diego Ongaro. bug in single-server membership changes, July 2015. URL `https://groups.google.com/g/raft-dev/c/t4xj6dJTP6E/m/d2D9LrWRza8J`.

Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference*, USENIXATC '14, pages 305–319, Berkeley, CA, USA, June 2014. USENIX Association. URL `https://dl.acm.org/doi/10.5555/2643634.2643666`.

Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety verification by interactive generalization. In Chandra Krintz and Emery Berger, editors, *Proceedings of the ACM SIGPLAN International Conference of Programming Language Design and Implementation*, PLDI '16, pages 614–630, New York, NY, USA, June 2016. Association for Computing Machinery. doi: https://doi.org/10.1145/2908080.2908118.

Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: Decidable reasoning about distributed protocols. *Proceedings of the ACM on Programming Languages*, OOPSLA, October 2017. doi: https://doi.org/10.1145/3140568.

Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Esteves-Verissimo. Velisarios: Byzantine fault-tolerant protocols powered by Coq. In Amal Ahmed, editor, *Proceedings of the European Symposium on Programming*, ESOP '18, pages 619–650, Berlin, Germany, June 2018. Springer. doi: https://doi.org/10.1007/978-3-319-89884-1_22.

Ganesan Ramalingam and Kapil Vaswani. Fault tolerance via idempotence. In *Proceedings of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 249–262, New York, NY, USA, January 2013. Association for Computing Machinery. doi: https://doi.org/10.1145/2429069.2429100.

Olivier Rütti, Zarko Milosevic, and André Schiper. Generic construction of consensus algorithms for benign and Byzantine faults. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 343–352, Washington, DC, USA, June 2010. IEEE Computer Society. doi: https://doi.org/10.1109/DSN.2010.5544299.

Denis Rystsov. CASPaxos: Replicated state machines without logs. *arXiv*, May 2018. doi: https://doi.org/10.48550/arXiv.1802.07000.

Fahad Saleh. Blockchain without waste: Proof-of-stake. *The Review of Financial Studies*, 34 (3):1156–1190, July 2020. doi: https://doi.org/10.1093/rfs/hhaa075.

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. RefinedC: Automating the foundational verification of C code with refined ownership types. In *Proceedings of the ACM SIGPLAN International Conference of Programming Language Design and Implementation*, PLDI '21, pages 158–174, New York, NY, USA, June 2021. Association for Computing Machinery. doi: https://doi.org/10.1145/3453483.3454036.

Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990. doi: https://doi.org/10.1145/98163.98167.

William Schultz, Ian Dardik, and Stavros Tripakis. Formal verification of a distributed dynamic reconfiguration protocol. In *Proceedings of the ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP '22, pages 143–152, New York, NY,

USA, January 2022a. Association for Computing Machinery. doi: https://doi.org/10.1145/3497775.3503688.

William Schultz, Siyuan Zhou, Ian Dardik, and Stavros Tripakis. Design and analysis of a logless dynamic reconfiguration protocol. In Quentin Bramas, Vincent Gramoli, and Alessia Milani, editors, *Proceedings of the International Conference of Principles of Distributed Systems*, OPODIS '21, Dagstuhl, Germany, February 2022b. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi: https://doi.org/10.4230/LIPIcs.OPODIS.2021.26.

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Proceedings of the ACM on Programming Languages*, POPL, December 2017. doi: https://doi.org/10.1145/3158116.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical Report RR-7506, INRIA, January 2011. URL `https://hal.inria.fr/inria-00555588`.

Ji-Yong Shin, Jieung Kim, Wolf Honoré, Hernán Vanzetto, Srihari Radhakrishnan, Mahesh Balakrishnan, and Zhong Shao. WormSpace: A modular foundation for simple, verifiable distributed systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, pages 299–311, New York, NY, USA, November 2019. Association for Computing Machinery. doi: https://doi.org/10.1145/3357223.3362739.

Victor Shoup. Practical threshold signatures. In Bart Preneel, editor, *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT '00, pages 207–220, Berlin, Germany, May 2000. Springer. doi: https://doi.org/10.1007/3-540-45539-6_15.

Vilhelm Sjöberg, Yuyang Sang, Shu chun Weng, and Zhong Shao. DeepSEA: A language for certified system software. *Proceedings of the ACM on Programming Languages*, OOPSLA, October 2019. doi: https://doi.org/10.1145/3360562.

Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, Inc., Hoboken, NJ, USA, 2 edition, 2006. URL `https://dl.acm.org/doi/10.5555/1202502`.

Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. Modularity for decidability: Implementing and semi-automatically verifying distributed systems. In *Proceedings of the ACM SIGPLAN International Conference of Programming Language Design and Implementation*, PLDI '18, pages 662–677, New York, NY, USA, June 2018. Association for Computing Machinery. doi: https://doi.org/10.1145/3192366.3192414.

Jeff Terrace and Michael J. Freedman. Object storage on CRAQ: High-throughput Chain Replication for read-mostly workloads. In *Proceedings of the USENIX Annual Technical Conference*, USENIXATC '09, Berkeley, CA, USA, June 2009. USENIX Association. URL `https://dl.acm.org/doi/10.5555/1855807.1855818`.

Ben Treynor. Today's outage for several Google services, January 2014. URL https://googleblog.blogspot.com/2014/01/todays-outage-for-several-google.html.

Klaus v. Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend synchrony: Synchronous verification of asynchronous distributed programs. *Proceedings of the ACM on Programming Languages*, POPL, January 2019. doi: https://doi.org/10.1145/3290372.

Robbert van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Computing Surveys*, 47(3), April 2015. doi: https://doi.org/10.1145/2673577.

Robbert van Renesse and Fred B. Schneider. Chain Replication for supporting high throughput and availability. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, OSDI '04, pages 91–104, Berkeley, CA, USA, December 2004. USENIX Association. URL https://dl.acm.org/doi/10.5555/1251254.1251261.

Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, November 1994. URL https://dl.acm.org/doi/10.5555/974938.

Chao Wang, Constantin Enea, Suha Orhun Mutluergil, and Gustavo Petri. Replication-aware linearizability. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the ACM SIGPLAN International Conference of Programming Language Design and Implementation*, PLDI '19, pages 980–993, New York, NY, USA, June 2019. Association for Computing Machinery. doi: https://doi.org/10.1145/3314221.3314617.

Michael Whittaker. CRAQ bug, June 2020. URL https://github.com/mwhittaker/craq_bug.

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the ACM SIGPLAN International Conference of Programming Language Design and Implementation*, PLDI '15, pages 357–368, New York, NY, USA, June 2015. Association for Computing Machinery. doi: https://doi.org/10.1145/2737924.2737958.

Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java system. In *Proceedings of the USENIX Conference on Object-Oriented Technologies*, COOTS '96, Berkeley, CA, USA, June 1996. USENIX Association. URL https://dl.acm.org/doi/10.5555/1268049.1268066.

Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the Raft consensus protocol. In *Proceedings of the ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP '16, pages 154–165, New York, NY, USA, January 2016. Association for Computing Machinery. doi: https://doi.org/10.1145/2854065.2854081.

Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 347–356, New York, NY, USA, July 2019. Association for Computing Machinery. doi: https://doi.org/10.1145/3293611.3331591.

Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '15, pages 263–278, New York, NY, USA, October 2015. Association for Computing Machinery. doi: https://doi.org/10.1145/2815400.2815404.

Appendix A

Additional ADO Examples

Some of the ADO examples in Section 4.4 intentionally make some simplifying assumptions (e.g., ignoring liveness or certain types of failures) for the sake of a cleaner presentation. This chapter shows that one can remove these assumptions to create objects that are closer to those used in real-world systems.

A.1 ADO Lock Alternatives

Different lock implementations have trade-offs in factors such as simplicity, fairness, and liveness. We have shown `CASLock`, which is very simple, but is lacking in the other two areas. Figure A.1 shows two other possible distributed lock ADOs that address these shortcomings.

`TicketLock` is a fairer, but slightly more complex alternative. It works by tracking the currently serving and next unused "tickets" as well as each replica's current ticket. A replica owns the lock when its ticket matches the one being served. The owner can then

```
ADO TicketLock {
  shared next : ℕ := 0;
  shared serving : ℕ := 0;
  shared tkts : Map[ℕ ⇀ ℕ] := {};
  method requestTkt() {
    if (this.nid ∈ this.tkts) {
      return this.tkts[this.nid];
    } else {
      this.next += 1;
      this.tkts[this.nid] := this.next;
      return this.next;
    }
  }
  method tryAcquire() {
    return this.nid ∈ this.tks && this.serving = this.tkts[this.nid];
  }
  method release() {
    if (this.nid ∈ this.tks && this.serving = this.tkts[this.nid]) {
      this.serving += 1;
      this.tkts.delete(this.nid);
    }
  }
}
```

```
ADO TimeoutLock {
  shared owner : option ℕ := None;
  shared time : ℕ := 0;
  method tryAcquire() {
    if (this.owner = None) {
      this.owner := Some(this.nid);
      this.time := 0;
    }
    return this.owner = Some(this.nid);
  }
  method release() {
    if (this.owner = Some(this.nid)) { this.owner := None; }
  }
  method checkin() {
    if (this.owner = Some(this.nid)) { this.time := 0; }
    return this.owner = Some(this.nid);
  }
  method tick() {
    this.time += 1;
    if (this.time > TIMEOUT) { this.owner := None; }
  }
}
```

Figure A.1: More complex ADO locks.

release the lock by incrementing the serving ticket, which causes the next waiting replica to acquire the lock. The waiting replicas implicitly form a queue, which provides better fairness than `CASLock` because replicas acquire the lock in the order they request it.

Unlike the standard concurrent implementation, in which clients of the lock maintain their own tickets, `TicketLock` remembers each replica's ticket in order to make `requestTkt` idempotent. For this lock, and indeed any non-preemptible distributed lock, the most significant difference with its concurrent counterpart is that it will block if the owner replica crashes because there is no way of revoking lock ownership. For simple examples and idealized models, this liveness limitation can be overlooked because it does not affect correctness, but practical settings require a solution. Production systems such as the Chubby distributed lock service [Burrows 2006] typically use a "keep-alive" approach where the lock is released if it does not hear from the client after some timeout. Although this is a simple solution for deadlock avoidance, it complicates the mutual exclusivity guarantee of the lock. One must either take special care to ensure that a client that dies and comes back online still believing to be the lock owner cannot interfere with the new owner, or else set the timeouts such that this situation is assumed to be impossible.

The ADO model has no notion of physical time, but one way of approximating a lock with a timeout (similar to Chubby) is `TimeoutLock`. It is essentially `CASLock` with an added `time` field that represents the elapsed time since the lock owner last renewed its lease by calling `checkin`. The `tick` method models the lock's internal clock by incrementing the timer and releasing the lock after a certain timeout period. In order for this to have some relation to physical time, this method must be continually called at regular intervals. This approach avoids the deadlock risk, but requires additional assumptions about clock skews

to maintain safety. If there were a sufficiently large delay between calling `checkin` and executing the critical section, it would be possible for another replica to acquire the lock and break mutual exclusivity. However, due to distributed systems' lack of a synchronized global clock, a common assumption is that the replica' local clocks are within some error bound so this situation cannot occur [Cachin et al. 2011; Hawblitzel et al. 2015a].

A.2 2PC with Recovery

The 2PC example in Section 4.4.2 improved the availability and reliability of the system by replicating the RMs so they can survive a certain number of crashes; however, it did not address the case where the TM crashes. If the TM crashes midway through a transaction, it is possible for the RMs to be in an inconsistent state where only some have received the request or final decision. Therefore, when a new TM comes online it must first perform a recovery operation before handling new requests.

Figure A.2 shows the same `TM` and `RM` from Figure 4.11, but augmented with a recovery phase (methods that have not changed from the previous example are elided). As before, the initialization begins by calling `pull` on every RM. Then the TM decides whether recovery is necessary by checking the latest transaction that each RM has received. If the transaction has not been decided (committed or aborted), then the TM must have crashed sometime during or before phase 2. If no RMs have undecided transactions, then no recovery is needed and the initialization ends. Otherwise, if some RM has a decision for a transaction that is undecided in another RM, the TM finishes replicating that decision in all RMs. Finally, the TM asks the RMs to delete any undecided transactions they received

```
ADO RM {
  shared txs : Vector[TX] := [];
  method prepare(tx) { ... }
  method decide(ts, decision) { ... }
  method remove_tx_after(ts) { this.txs := this.txs.filter(λ tx. tx.ts ≤ ts); }
  method read() { return this.txs; }
}

DApp TM(rm_1: RM, ..., rm_n: RM) {
  local ts : ℤ := 0;
  /* Must be called once when TM starts */
  proc init() {
    for rm in [this.rm_1, ..., this.rm_n] {
      while (rm.pull() = FAIL) {}
    }
    /* Recovery */
    need_recovery := [];
    /* Check the last decided TX in each RM */
    for rm in [this.rm_1, ..., this.rm_n] {
      rm.invoke(read());
      do { txs := rm.push(); } while (txs = FAIL);
      need_recovery[rm] := txs.last().decision ∉ [COMMIT, ABORT];
      last_decided := txs.filter(λ tx. tx.decision ∈ [COMMIT, ABORT]).last();
      /* Remember the timestamp and decision of the latest decided TX */
      if (last_decided.ts > this.ts) {
        this.ts := last_decided.ts;
        decision := last_decided.decision;
      }
    }
    /* All RMs are in consistent state */
    if (!need_recovery.any()) { return; }
    for rm in [this.rm_1, ..., this.rm_n] {
      /* Finish TXs that failed during phase 2 */
      if (need_recovery[rm]) { rm.invoke(decide(this.ts, decision)); }
      /* Wipe out TXs that failed during phase 1 */
      rm.invoke(remove_tx_after(this.ts));
      while (rm.push() = FAIL) {}
    }
  }
  proc handle_request(ops) { ... }
}
```

Figure A.2: Two-Phase Commit with recovery.

after the latest decided one. After ensuring that the methods are complete by calling push, the RMs are guaranteed to be in a consistent state again and the TM can begin handling new transaction requests.

Appendix B

Additional Refinement Details

The key component of a refinement between a network-based specification and the ADO model is the relation between a replica's local log and its active branch in the cache tree (Figure B.1). Intuitively, this says that for every log, there is always a branch that can be transformed into the same log by collecting the *MCaches* along the branch, and furthermore, the branch is at least as recent as the replica's latest supported *CCache* (i.e., the log contains all of the replica's committed methods).

Each of the refinement proofs discussed for the different instances of the ADO model proves this relation, albeit with minor variations (e.g., Adore also includes *RCaches* in *branchToLog*, AdoB weakens *LogMatch* to only hold for honest replicas), but there are

$$
\begin{aligned}
branchToLog(tr, C) &\triangleq \{(t, M) \mid MCache(\_, t, M) \in ancestors(tr, C)\} \\
LogMatch(st_{net}, st_{ADO}) &\triangleq \forall nid.\, \exists C.\, log(st_{net}, nid) = branchToLog(tree(st_{ADO}), C) \\
&\quad \wedge C \succeq activeC(tree(st_{ADO}), nid)
\end{aligned}
$$

Figure B.1: The general local log-ADO branch correspondence of $\mathbb{R}$.

small differences in their approaches as well. The following sections provide additional information and definitions about these differences.

B.1 Advert Refinement Details

Definition 11 (Phase Scheduler)**.** *A* phase schedule *is a list of Elect and Commit events indicating the order that a certain replica will call the corresponding operations. A* phase scheduler, $\mathbb{O}_{sched}$*, is an oracle that takes the current network state and a replica's node ID and returns a phase schedule for that replica.*

Definition 12 (Complete Request)**.** *A request is* complete *if there is a ghost Begin message, and a quorum of replicas have received and either accepted or rejected the request. Otherwise the request is* incomplete*.*

Definition 13 (Completion)**.** *Given a network state with an incomplete request, a* completion *of that request can be computed by repeatedly querying a phase scheduler and applying either* `elect` *or* `commit` *as appropriate until the request is complete.*

Definition 14 (Logical Time Reordering)**.** *The* logical time reordering *of a trace of network events is computed by sorting the events by their logical times with the following caveats: events originating from the same replica remain in program order, messages cannot be delivered before they are sent, and once a replica receives a message with logical time t, all later messages at the same replica with logical time less than t are dropped.*

The refinement relation ($\mathbb{R}_{\text{Advert}}$) is proved by considering an arbitrary network state and showing that an ADO state exists that satisfies *LogMatch*. This is done by computing

the completion and logical time reordering of the network and proving that the resulting state is equivalent to the original in the sense that the local logs are the same. Then, a related ADO state can be constructed from this network state by applying `pull` for every successful `elect`, `invoke` for every local log update, and `push` for every successful `commit`.

B.2 Adore Refinement Details

B.2.1 SRaft and Adore

The following proof sketch demonstrates the intuition for why the *LogMatch* relation is preserved between SRaft and Adore states.

Lemma 7 (Adore Refinement)**.** *Suppose LogMatch holds for some SRaft state and a cache tree, tr, and that replica S's local log is λ. For any valid SRaft step where S's new log is λ' there is a valid Adore step to some tr' with a cache C such that $branchToLog(tr', C) = \lambda'$.*

Proof Sketch. Consider each possible SRaft operation. Neither `elect` nor `commit` change λ, so Adore can take a stutter step and the result holds trivially. Both `invoke` and `reconfig` append a new method to λ, and the corresponding Adore operations append an equivalent cache to the end of S's active branch, preserving the relation. If `deliver`'s message is an election request or acknowledgement, then λ does not change and, likewise, `pull` only adds an *ECache* to the cache tree, which *branchToLog* ignores. The only other operation to change λ is a delivery of a commit request, in which case S takes the log λ' sent by some leader L. `push` adds a *CCache*, C to L's branch, and, because *branchToLog* also ignores *CCaches*, we know by induction that $branchToLog(tr', C) = branchToLog(tr', parent(C)) =$

$$\mathbb{R}_{\text{net}}(st, st') \triangleq \forall nid.\, log(st, nid) = log(st', nid) \wedge time(st, nid) = time(st', nid)$$

Figure B.2: The network-equivalence relation, $\mathbb{R}_{\text{net}}$.

$log(st_{net}, L)$. Therefore, because S voted for L's commit, their logs are equal so $\lambda' = log(st_{net}, L) = branchToLog(tr', C)$ and *LogMatch* still holds. □

B.2.2 Raft and SRaft

Recall that SRaft is essentially the same as Raft, aside from some simplifying assumptions about how and when messages are delivered. To prove that Raft refines SRaft, we must show that, despite these assumptions, SRaft behaves equivalently to Raft, up to $\mathbb{R}_{\text{net}}$ (Figure B.2), which states that the relevant parts of every replica's local state is the same. The following proof sketches explain the intuition for why each assumption is safe.

Definition 15 (Valid Message)**.** *Upon receiving a message, every replica checks that it satisfies certain properties, and ignores it if it does not. Examples include a request with an insufficiently large timestamp, or an acknowledgement for a request that has already ended. A message is* valid *if it satisfies the properties and is therefore not ignored.*

Lemma 8 (SRaft Valid Messages)**.** *For any sequence of network events, evs, that results in a state, st, there exists a sequence of events, evs′, that results in an equivalent state, st′ ($\mathbb{R}_{\text{net}}(st, st')$), such that evs′ contains only valid messages.*

Proof Sketch. It is trivial to define *evs′* by filtering out invalid messages from *evs*. Since invalid messages are ignored anyway, this has no effect on the replicas' local states. □

Definition 16 (Ordered Messages)**.** *Messages m and m′ are ordered if*

$$(time(m), vrsn(m)) \leq_{lex} (time(m'), vrsn(m'))$$

where $\leq_{lex}$ is the usual lexicographic order.

Definition 17 (Locally Ordered Messages)**.** *A sequence of network events, evs, is locally ordered if, for every m and m′, such that evs = . . . • deliver(m) • . . . • deliver(m′) • . . . and to(m) = to(m′), m and m′ are ordered.*

Definition 18 (Globally Ordered Message)**.** *A sequence of network events, evs, is globally ordered if, for every m and m′, such that evs = . . . • deliver(m) • . . . • deliver(m′) • . . ., m and m′ are ordered.*

Lemma 9 (SRaft Globally Ordered)**.** *For any sequence of network events, evs, that contains only valid messages that results in a state, st, there exists a sequence of events, evs′, that results in an equivalent state, st′ ($\mathbb{R}_{\text{net}}(st, st')$), such that evs′ is globally ordered.*

Proof Sketch. We know *evs* is already locally ordered because non-locally ordered messages would be ignored, and there are no invalid messages by assumption. Therefore, all that remains is to show that sorting the globally unordered messages in *evs* does not affect the replicas' local states. Observe that receiving a message is a local operation in that it only affects the state of the recipient. Therefore, deliveries to different recipients are independent and can freely commute. Since we have established that messages are locally ordered, the only out-of-order messages must have different recipients and can be sorted without affecting the final global state. □

Definition 19 (Atomic Deliveries)**.** *An operation in a sequence of network events, evs, is*

delivered atomically if every corresponding delivery (both of the request and the acknowledgements) is adjacent in evs.

Lemma 10 (SRaft Atomic)**.** *For any sequence of network events, evs, that contains only globally ordered, valid messages that results in a state, st, there exists a sequence of events, evs′, that results in an equivalent state, st′ ($\mathbb{R}_{\text{net}}(st, st')$), such that evs′ has atomic deliveries.*

Proof Sketch. To construct *evs′*, for every operation, we must find all unadjacent corresponding deliveries and "push" them together in such a way that does not affect the resulting state. Because *evs* is globally ordered, and no leader uses the same timestamp-version number pair for its operations, any messages that come between two related deliveries must originate from a different leader. This also implies that the deliveries must have different recipients, because a replica would not accept two requests from different leaders with the same timestamp. Therefore, these delivery events can commute. By repeating this process, all delivery events can be rearranged so that corresponding ones are adjacent, and one can treat them as if they occurred atomically. □

Lemma 11 (Raft Refines SRaft)**.** *For any sequence of network events, evs, that results in a state, st, there exists a sequence of events, evs′, that results in an equivalent state, st′ ($\mathbb{R}_{\text{net}}(st, st')$), such that evs′ contains only valid, globally ordered, and atomic messages.*

Proof Sketch. Trivial combination of Lemmas 8 to 10. □

Theorem 5 (Raft Refines Adore)**.** *Suppose $\mathbb{R}$ holds for some Raft state and a cache tree, tr, and that replica S's local log is λ. For any valid Raft step where S's new log is λ' there is a valid Adore step to some tr' with a cache, C, such that $branchToLog(tr', C) = \lambda'$.*

Proof Sketch. Trivial combination of Lemmas 7 and 11. □

B.3 AdoB Refinement Details

This section summarizes each of the 15 conjuncts that make up $\mathbb{R}$.

LogMatch Every replica's local log corresponds to some branch in the cache tree; i.e., every log entry has an equivalent *MCache*. Furthermore, for an honest replica, the corresponding branch is at least as recent as its *activeC*. This allows us to prove that AdoB's safety and liveness implies GenJolteon's safety and liveness.

RequestLogMatch The log contained in every `invoke` and `commit` request corresponds to some branch of the cache tree. This is used to prove *LogMatch* is maintained after `invoke` or `commit`.

ElectLogMatch The logs contained in every `elect` message corresponds to some branch of the cache tree. Furthermore, the latest log is a *CCache* or *TCache* from the previous round. This is used to prove *LogMatch* is maintained after `elect`.

TimeoutLogMatch The logs contained in every `timeout` message corresponds to some branch of the cache tree. If the timeout occurs after a successful commit, then the log corresponds to the newly created *CCache*. Otherwise, there is no *CCache* for the current round, but the latest log is still at least as recent as the latest *CCache*. This is used to prove *LogMatch* is maintained after `timeout`.

ReceivedLogSent Replicas sometimes store multiple logs from other replicas, for example when it receives an election or timeout message. Each of these logs must have been carried by some request in the network event history. This is used to prove *LogMatch* is maintained when these logs are sent along with an `invoke` request.

LocalTimeMatch An honest replica's local time in the network model must match its local time in AdoB and there must be a *CCache* or *TCache* with the same time. The time in the network model may also be one greater than the AdoB time if a replica has just committed a method. This is used to prove that the preconditions for `pull`, `invoke`, `push`, and `timeout` that reference local times are satisfied in both models.

CacheTimeMatch For every cache with a timestamp t, there exists a quorum of honest replicas whose local time is at least t. This is used to rule out the existence of caches with very high timestamps.

VotedTimeMatch An honest replica must either have a larger timestamp than its latest voted cache, or they are equal and the replica's phase dictates the type of the voted cache. If the replica is invoking a method or has just voted for `invoke`, the cache must be anything but an *ECache*. If it is committing or has just voted for `commit`, the voted cache must be a *CCache* or a *TCache*. This is used to ensure that a replica's phase is somewhat synchronized with what caches it has voted for.

PreviousRoundTimeMatch If an honest replica has timestamp $t > 1$, there exists a *CCache* or *TCache* with time $t - 1$. This is used to ensure the rounds in the network and AdoB models stay synchronized.

CommitAckDelivered Every *CCache* corresponds to a delivered `commit` acknowledgement. This is used to support *RequestLogMatch*.

CommitRequestDelivered Every *CCache* corresponds to a delivered `commit` request. This is used to support *RequestLogMatch*.

TimeoutDelivered Every *TCache* corresponds to a delivered `timeout` message. This is used to support *TimeoutLogMatch*.

TimeoutUnique The timestamp of every `timeout` message is unique. This is used in conjunction with *TimeoutDelivered* to support *TimeoutLogMatch*.

TimeoutDone If an honest replica has received a `timeout` message and has not yet moved to a later round, its phase must be *Done*, which prevents it from acknowledging any further requests in that round. This is used to ensure a `timeout` effectively ends a round.

ActiveTCache If an honest replica's active cache is a *TCache*, then its local timestamp must be greater than the *TCache*'s. This is used to ensure that replicas advance their timestamps after timing out.

Appendix C

Additional Safety Proof Details

C.1 Advert Safety Proof Details

This section explains the high-level structure of the replicated state safety proof template for Paxos-like protocols and shows a few representative theorems. Figure C.1 shows an overview of the dependencies between theorems in the template. Theorems are categorized into *protocol invariants*, which are properties that must be proven for each instantiation (e.g., Multi Paxos, Vertical Paxos, etc), and *core theorems*, which hold for the whole family of protocols and are proven once and for all.

Core Theorems The following are important safety properties of replicated distributed systems and their supporting theorems. Aside from the protocol invariants, they make no assumptions about the implementation or lower-level details and hold for most Paxos variants.

To prove replicated state safety, we must show that every replica's local *rdata* snapshot

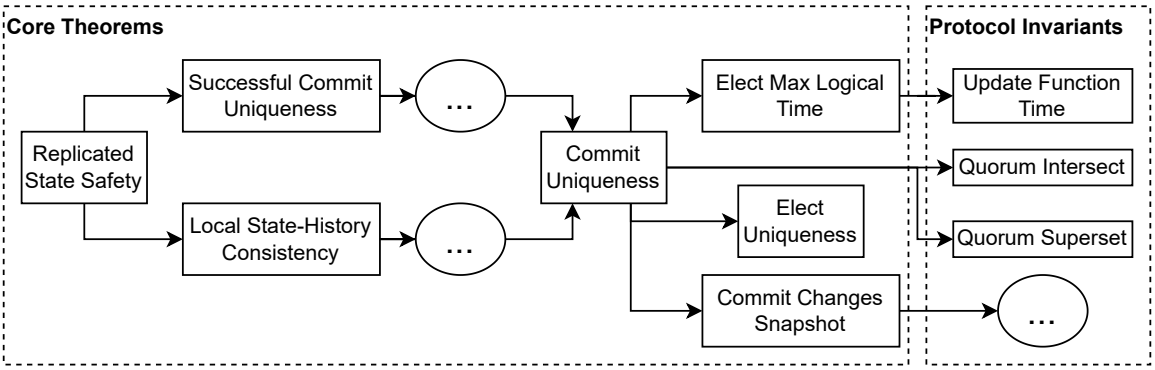


Figure C.1: Abridged proof dependencies.

is associated with a successful *Commit* operation, and that *Commit* operations have unique timestamps. Combined with the fact that *Commits* require approval from a quorum of replicas, and that replicas only accept *Commits* in strictly increasing timestamp order, this guarantees that there is exactly one linearized order in which the methods that created the current *rdata* could be committed.

Core Theorem 1 (Local State-History Consistency)**.** *For every replica's rdata snapshot, there exists a corresponding successful Commit message in the network history. A successful Commit message is one that was accepted by a quorum of servers.*

Core Theorem 2 (Successful Commit Uniqueness)**.** *Any two successful Commit messages associated with the same logical time contain the same rdata.*

Proof Sketch. We proceed by induction on the network history. Because logical times are unique (Core Lemma 1), the two *Commit* messages must come from the same replica. Then, because a *Commit* message must follow an *Elect* request, there are two cases: an election occurred between the commits, or there is an intermediate third commit. In the first case, we are done because elections change the logical time (Core Lemma 4), which contradicts the assumption that they are equal. In the second case, we use the inductive hypothesis

to show that the first and intermediate *Commit* messages send the same *rdata* and then again derive a contradiction from the change in *rdata* between the intermediate and the last *Commit*. □

Core Theorem 2 expresses a kind of immutability; once an *rdata* snapshot is committed at a particular logical time, it can never be erased or overwritten. For protocols with physical logs (e.g., Multi Paxos), this implies immutability of each index of the log, but even for protocols with in-place updates to a single object (e.g., CASPaxos), this represents the fact that there exists an unchanging logical history of atomic updates.

Core Theorems 1 and 2 depend on the following core lemmas.

Core Lemma 1 (Commit Uniqueness)**.** *No two Commit messages have the same logical time.*

Core Lemma 2 (Commit Changes Snapshot)**.** *A replica's local rdata snapshot changes only at the end of an election (when it is copied from servers) or at the end of a commit operation (when a new update is appended to it).*

Core Lemma 3 (Elect Uniqueness)**.** *No two Commit messages have the same logical time.*

Core Lemma 4 (Elect Max Logical Time)**.** *The latest logical time in the corresponding rdata after a successful election is the maximum among the participating voters.*

Core Lemma 1 guarantees that either the time must change between commit phases. Core Lemma 2 describes when and how a client's state changes. Core Lemmas 3 and 4 guarantee basic properties about logical timestamps during elections.

Protocol Invariants Whereas many of the core theorems require nuanced reasoning about concurrent, asynchronous behaviors, the protocol invariants are intended to be simple properties that follow directly from a protocol's instantiations of the parameters. A few such properties are listed below to give their flavor.

Protocol Invariant 1 (Quorum Intersect)**.** *If S_1 and S_2 are quorums (with respect to isQuorum) for Elect and Commit phases in the same logical time respectively, then the two sets have a non-empty intersection.*

Protocol Invariant 2 (Quorum Superset)**.** *If S is a quorum and $S \subseteq S'$, then S' is also a quorum.*

Protocol Invariant 3 (Update Function Time)**.** *Extracting the time from an rdata with rtime after applying the update function (update) must return the same timestamp passed to the update function.*

Protocol Invariants 1 and 2 define properties of quorum-based consensus protocols and are straightforward to prove with set-theoretic arguments for most reasonable implementations of *isQuorum*. Protocol Invariant 3 enforces a relation between the time used by *update* and returned by *rtime*.

C.2 Adore Safety Proof Details

Lemma 12 (Descendant Order)**.** *If C_Y is a descendant of C_X then $C_Y \succ C_X$.*

Proof Sketch. It is enough to show that every newly added cache is greater than its parent. An *ECache* added by `pull` has a larger timestamp than its supporters, including its parent.

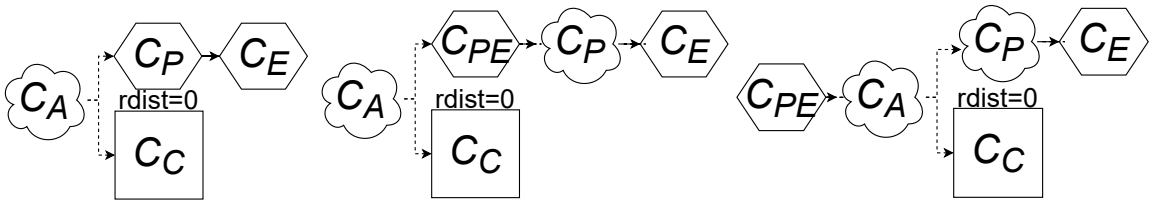
MCaches and *RCaches* increment their parent's version number. `push` copies the parent's time and version, but since the parent must be an *MCache* or *RCache*, the *CCache* is greater by definition of $\succ$. □

Lemma 13 (Leader Uniqueness, rdist-0)**.** *If C_{E1} and C_{E2} are ECaches and $rdist(C_{E1}, C_{E2}) = 0$, then $time(C_{E1}) \neq time(C_{E2})$.*

Proof Sketch. Because $rdist = 0$, C_{E1} and C_{E2} have the same configuration and thus overlapping quorums of supporters. `pull` chooses a time greater than any observed by its supporters, so, since C_{E1} and C_{E2} share a supporter, whichever cache was added to the tree last must have a larger timestamp. □

Theorem 6 (Election-Commit Order, rdist-0)**.** *Let C_C be a CCache and C_E be an ECache such that $C_E \succ C_C$ and $rdist(C_E, C_C) = 0$. C_E must be a descendant of C_C.*

Proof Sketch. If C_E is a descendant of C_C then we are done, so suppose it is not for the sake of deriving a contradiction. By Lemma 12, C_C cannot be a descendant of C_E either. By double induction on the time and version number, we can assume that C_E is the first *ECache* that is not a descendant of C_C and for all *ECaches* C'_E where $C_E \succ C'_E \succ C_C$, C'_E is a descendant of C_C. Because $rdist = 0$, C_E and C_C have the same configuration and thus overlapping quorums of supporters. Therefore, C_E's parent, C_P, is greater than C_C because `pull` selects the largest cache supported by its supporters. This leaves the three following options for the shape of the tree.

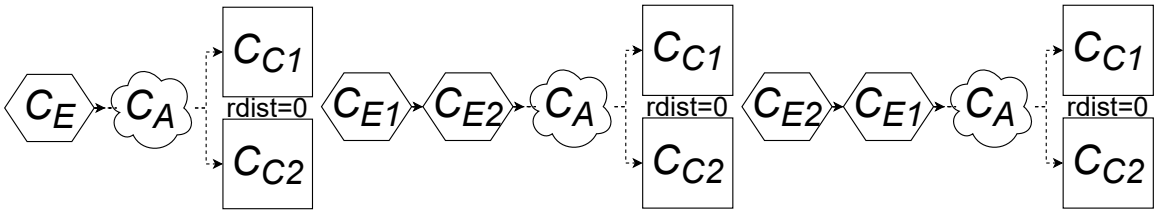


In the first case, C_P is an *ECache*, but then $C_E \succ C_P \succ C_C$, so by the inductive hypothesis it must be a descendant of C_C, which is a contradiction. If C_P is not an *ECache*, it must have an *ECache* ancestor, C_{PE}, such that $time(C_P) = time(C_{PE})$. From $C_P \succ C_C$, we know $time(C_P) \geq time(C_C)$, but, thanks to Lemma 13, we also know that if $time(C_P) = time(C_C)$ they must be on the same branch. Since they are not, $time(C_P) = time(C_{PE}) > time(C_C)$, so $C_{PE} \succ C_C$.

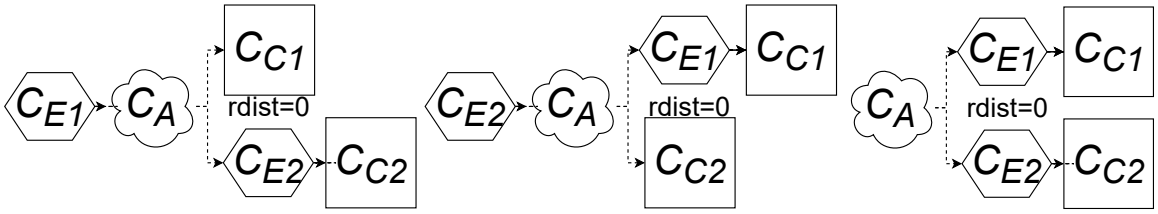
The two possible locations for C_{PE} are between the common ancestor C_A and C_P, or before C_A. The first option derives a contradiction because $C_E \succ C_{PE} \succ C_C$ but C_{PE} is not a descendant of C_C. The second case is also impossible by Lemma 12 because C_{PE} is an ancestor of C_C but $C_{PE} \succ C_C$. Thus, it is impossible to arrive at a cache tree in which C_E is not a descendant of C_C. □

Theorem 7 (Safety, rdist-0)**.** *Let C_{C1} and C_{C2} be CCaches such that $rdist(C_{C1}, C_{C2}) = 0$. Either C_{C1} is a descendant of C_{C2} or C_{C2} is a descendant of C_{C1}.*

Proof Sketch. As before, assume that neither C_{C1} nor C_{C2} is a descendant of the other in order to derive a contradiction. Each must have a nearest *ECache* ancestor, C_{E1} and C_{E2} respectively, with the same time and no intervening *ECaches*. Three possibilities are as follows.



In the first case, C_{E1} and C_{E2} are the same *ECache*, called C_E, and, in the other two, the elections are distinct common ancestors of C_{C1} and C_{C2}. Recall that there can be no *ECaches* between C_{E1} and C_{C1} and likewise for C_{E2} and C_{C2}, so the latter two cases are impossible. Similarly, in the first case, there are no *ECaches* anywhere on the forking branches after C_A, but this is also impossible because `pull` is the only operation that can create forks in the tree. This leaves three options.



In the first case, C_{E2} is a descendant of C_{E1} so $C_{E2} \succ C_{E1}$ by Lemma 12. This also implies $time(C_{E2}) > time(C_{E1})$ because an *ECache*'s version number is always 0. Then, because $time(C_{E1}) = time(C_{C1})$, $C_{E2} \succ C_{C1}$ as well. By Theorem 6, C_{E2} must be a descendant of C_{C1}, but it is not, so this case is impossible. The second case follows by a symmetric argument. The final case also contradicts Theorem 6 if $time(C_{E1}) \neq time(C_{E2})$, which we know is true by Lemma 13. □

Lemma 14 (Leader Uniqueness, rdist-1)**.** *If C_{E1} and C_{E2} are ECaches and $rdist(C_{E1}, C_{E2}) = 1$, then $time(C_{E1}) \neq time(C_{E2})$.*

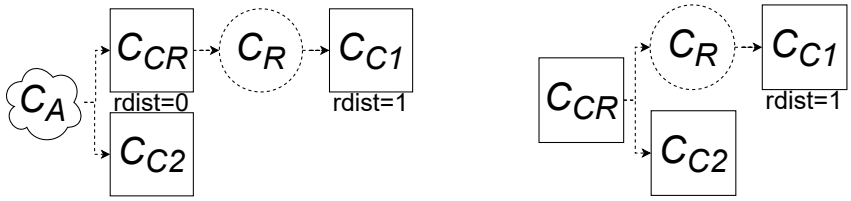
Proof Sketch. R1$^+$ guarantees that quorums of C_{E1} and C_{E2}'s configurations overlap. `pull` chooses a time greater than any observed by its supporters, so, since C_{E1} and C_{E2} share a supporter, whichever cache was added to the tree last must have a larger timestamp. □

Theorem 8 (Election-Commit Order, rdist-1)**.** *Let C_C be a CCache and C_E be an ECache such that $C_E > C_C$ and $rdist(C_E, C_C) = 1$. C_E must be a descendant of C_C.*

Proof Sketch. The proof of Theorem 6 relied on Lemma 12 and Lemma 13 to show that every case where C_E is not a descendant of C_C is impossible. The only difference in this case is that $rdist = 1$; however, Lemma 12 is independent of *rdist*, and Lemma 13 can be replaced by Lemma 14, so nearly exactly the same proof as before works. □

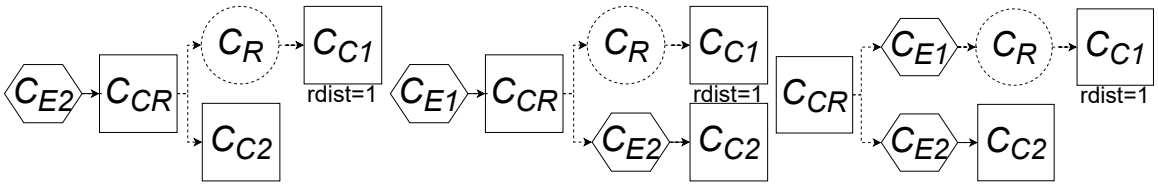
Theorem 9 (Safety, rdist-1)**.** *Let C_{C1} and C_{C2} be CCaches such that $rdist(C_{C1}, C_{C2}) = 1$. Either C_{C1} is a descendant of C_{C2} or C_{C2} is a descendant of C_{C1}.*

Proof Sketch. If C_{C1} and C_{C2} are on the same branch, then we are done, so suppose they are not for the sake of deriving a contradiction and let C_A be a common ancestor. Because $rdist = 1$, there is one *RCache*, C_R, between either C_A and C_{C1} or C_A and C_{C2}. Without loss of generality, suppose it is on C_{C1}'s branch. By R3, C_R must have a *CCache* ancestor, C_{CR}, with the same time and no other intervening *CCaches*. The possible locations for C_{CR} are the following.



The first case has two *CCaches* with $rdist = 0$ on separate branches, which is impossible

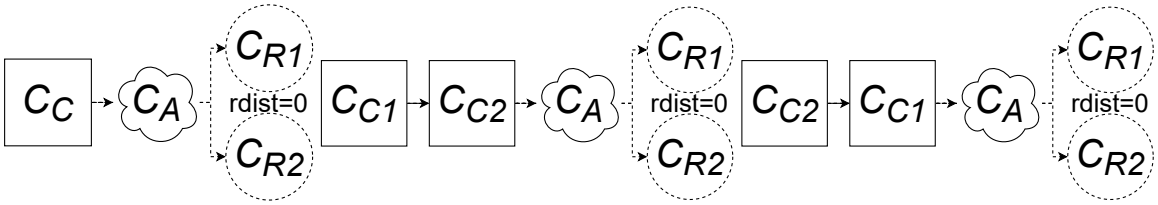
by Theorem 7. In the second case, C_{CR} is a common ancestor of C_R and C_{C2}. Each *CCache* must have a nearest *ECache* ancestor, C_{E1} and C_{E2}, respectively, with the same time and no intervening *ECaches*. Three possibilities are as follows.



In the first case, recall that $time(C_R) = time(C_{CR})$, which implies that there are no *ECaches* between them. Likewise, there are none between C_{E2} and C_{C2}, so neither fork has an *ECache*, which is impossible. In the second case, $C_{E2} \succ C_{E1}$ by Lemma 12 so $time(C_{E2}) > time(C_{E1})$. Then, since $time(C_{E1}) = time(C_R) = time(C_{CR})$, $C_{E2} \succ C_{CR}$ as well. But this contradicts Theorem 8 since C_{E2} is not a descendant of C_{CR}. The final case also contradicts Theorem 8 because $time(C_{E1}) \neq time(C_{E2})$ by Lemma 14, so one must be greater than the other. This leaves no option but that C_{C1} is a descendant of C_{C2} or vice-versa. □

Lemma 15 (CCache in RCache Fork)**.** *Let C_{R1}, C_{R2} be RCaches such that $rdist(C_{R1}, C_{R2}) = 0$, and neither is a descendant of the other, but both have a common ancestor C_A. Then there exists a CCache C_C that is a descendant of C_A and an ancestor of either C_{R1} or C_{R2}.*

Proof Sketch. By R3, each *RCache* must have a *CCache* ancestor with the same time, called C_{C1} and C_{C2} respectively. If either C_{C1} or C_{C2} is a descendant of C_A, then we are done. Otherwise, the options are for C_{C1} and C_{C2} to be the same cache, or for one to be a descendant of the other.

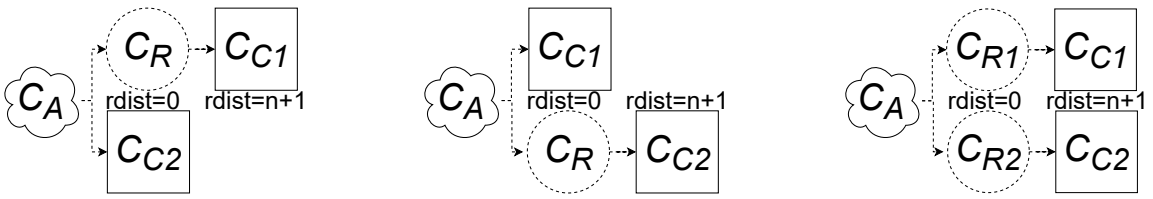


Recall that there cannot be any *ECaches* between C_{C1} and C_{R1} or between C_{C2} and C_{R2}. This means that, in every case, that are no *ECaches* on either forking branch, which is impossible. □

Theorem 10 (Safety)**.** *Let tr be a cache tree with any rdist. For any CCaches C_{C1} and C_{C2} in tr, one must be a descendant of the other.*

Proof Sketch. We proceed by induction on $rdist(tr)$. For $rdist \leq 1$, we are done by Theorems 7 and 9. Suppose now that all trees with $rdist = n$ are safe, and $1 < rdist(tr) = n + 1$ so $1 < rdist(C_{C1}, C_{C2}) \leq n + 1$. If $rdist(C_{C1}, C_{C2}) \leq n$, then they are in some subtree tr' with $rdist(tr') = n$, so we are done by the inductive hypothesis. Safety also holds if C_{C1} and C_{C2} are on the same branch, and, if not, we will show that all other shapes for tr are impossible.

There are two options for how the n *RCaches* could be distributed. Either all $n + 1$ *RCaches* are on one branch (e.g., $rdist(C_A, C_{C1}) = n + 1$ and $rdist(C_A, C_{C2}) = 0$), or the *RCaches* are distributed such that there is at least one on both branches. In either case, we can identify the first *RCache* descendant of C_A on both branches.



The first two cases are symmetric, so assume without loss of generality that C_R is on C_{C1}'s branch. Let C_{CR} be the first *CCache* descendant of C_R. It is enough to show that $rdist(C_{CR}, C_{C2}) \leq n$ because then C_{CR} and C_{C2} must be on the same branch, which is a contradiction. We know $rdist(C_{C1}, C_{C2}) > 1$, so C_R cannot be the only *RCache* on C_{C1}'s branch. We also know by R2 that this other *RCache* cannot be between C_R and C_{CR}. Therefore, this *RCache* does not count towards $rdist(C_{CR}, C_{C2})$ and it is at most n.

For the final case, by Lemma 15 there must be a *CCache* between C_A and either C_{R1} or C_{R2}. Suppose it is between C_A and C_{R2} and call it C'_{C2}. Now $rdist(C_{R1}, C'_{C2}) = 0$, so this is the same case as before. Therefore, this situation is impossible as well and the only possibility is that C_{C1} and C_{C2} are on the same branch. □

C.3 AdoB Safety and Liveness Proof Details

This section contains Coq formalizations of certain key definitions and theorems for the safety and liveness of AdoB.

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].
```

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).
```

The global time is always guaranteed to eventually increase.

```
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'.
```

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.
```

The parent cache chosen by $\mathbb{O}_{push}$ is the most recent of the leader's *MCaches*.

```
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 ->
    cm ≥ c.
```