Much ADO about Failures: A Fault-Aware Model for Compositional Verification of Strongly Consistent Distributed Systems*

WOLF HONORÉ[†], Yale University, USA
JIEUNG KIM[†‡], Yale University, USA
JI-YONG SHIN, Northeastern University, USA
ZHONG SHAO, Yale University, USA

Despite recent advances, guaranteeing the correctness of large-scale distributed applications without compromising performance remains a challenging problem. Network and node failures are inevitable and, for some applications, careful control over how they are handled is essential. Unfortunately, existing approaches either completely hide these failures behind an atomic state machine replication (SMR) interface, or expose all of the network-level details, sacrificing atomicity. We propose a novel, compositional, atomic distributed object (ADO) model for strongly consistent distributed systems that combines the best of both options. The object-oriented API abstracts over protocol-specific details and decouples high-level correctness reasoning from implementation choices. At the same time, it intentionally exposes an abstract view of certain key distributed failure cases, thus allowing for more fine-grained control over them than SMR-like models. We demonstrate that proving properties even of composite distributed systems can be straightforward with our Coq verification framework, Advert, thanks to the ADO model. We also show that a variety of common protocols including multi-Paxos and Chain Replication refine the ADO semantics, which allows one to freely choose among them for an application's implementation without modifying ADO-level correctness proofs.

CCS Concepts: • **Software and its engineering** → **Software verification**; **Distributed programming languages**; • **Theory of computation** → **Distributed computing models**; *Object oriented constructs.*

Additional Key Words and Phrases: distributed systems, formal verification, proof assistants, Coq

ACM Reference Format:
Wolf Honoré, Jieung Kim, Ji-Yong Shin, and Zhong Shao. 2021. Much ADO about Failures: A Fault-Aware Model for Compositional Verification of Strongly Consistent Distributed Systems. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 97 (October 2021), 42 pages. https://doi.org/10.1145/3485474

1 INTRODUCTION

It is difficult to guarantee correctness and efficiency of a distributed systems at the same time because even the simplest distributed systems employ sophisticated protocols to coordinate failure-prone nodes over an unreliable network. To further complicate matters, modern applications are

*The present version, published by the authors as Yale University Technical Report YALEU/DCS/TR-1557 [Honoré et al. 2021] includes supplementary appendices after the References section.
[†]Both authors contributed equally to this research.
[‡]Jieung Kim is now at Google Research.

Authors' addresses: Wolf Honoré, wolf.honore@yale.edu, Yale University, USA; Jieung Kim, jieungkim@google.com, Yale University, USA; Ji-Yong Shin, j.shin@northeastern.edu, Northeastern University, USA; Zhong Shao, zhong.shao@yale.edu, Yale University, USA.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2021 Copyright held by the owner/author(s).
2475-1421/2021/10-ART97
https://doi.org/10.1145/3485474

often built from a combination of distributed systems [Dean 2009], which necessitates reasoning about interleaving and concurrent interactions between them.

Distributed systems offer varying reliability and consistency guarantees [Tanenbaum and van Steen 2006], but in this paper we focus on strongly consistent protocols, including consensus (e.g., multi-Paxos [Renesse and Altinbuken 2015], and Raft [Ongaro and Ousterhout 2014]), and some instances of primary backup (e.g., Chain Replication [Renesse and Schneider 2004]) because even weakly consistent systems rely on them to handle critical operations. Unless otherwise specified, we use the term "distributed system" to refer to this class of protocols.

These protocols are specified in terms of operations in an asynchronous, unreliable network and can be quite complex [Boichat et al. 2003; Lamport 2001; Lampson 2001; Renesse and Altinbuken 2015]. Recognizing that this low-level network-based interface is not well suited for building and reasoning about distributed applications, developers often use higher-level abstractions such as file systems [MacCormick et al. 2004], databases [Chang et al. 2006], and state machine replication (SMR) [Schneider 1990]. These greatly simplify application-level reasoning by hiding internal details (e.g., the transient, intermediate states that arise from network failures) behind an atomic interface. However, this black-box approach makes it impossible to express or reason about systems with optimizations that "open up" the underlying protocols and rely on such details [Gray and Lamport 2006; Zhang et al. 2015].

Because bugs in distributed systems can cause critical failures [The AWS Team 2011; Treynor 2011] and exhaustive testing is often impractical, formal verification is required to be sure of an application's correctness. In existing verification frameworks for distributed systems [Hawblitzel et al. 2015a; Krogh-Jespersen et al. 2020; Sergey et al. 2017; Wilcox et al. 2015], developers either write applications in terms of asynchronous network events and prove that they are equivalent to some atomic specification, or they use an atomic SMR-like interface built on an existing verified consensus protocol such as multi-Paxos or Raft. The former option offers much more flexibility in terms of implementation and optimization than the latter, which is limited to the choices made by the verified consensus protocol. On the other hand, network-based specifications blend application- and protocol-level reasoning (e.g., proving that a distributed queue behaves correctly vs. proving the linearizability of Paxos), which complicates both, and reduces the generality of the specifications and proofs by closely tying them to a specific implementation.

In this paper we present the novel *atomic distributed object* (ADO) model as a happy medium between the simplicity of SMR and the expressive power of network models. The ADO model defines an atomic semantics that faithfully captures the common high-level behaviors of strongly consistent distributed protocols, including important cases such as nondeterministic failures and transient states, while abstracting away irrelevant protocol-specific details such as packet interleaving and quorum sizes. This allows an application to swap its implementation for any strongly consistent protocol without modifying its ADO-level specification or proofs. Protocols with standard network-based specifications can also be shown to implement the ADO model through contextual refinement [Filipovic et al. 2010; Liang et al. 2013], which means that properties of the high-level model carry down to the implementation. ADO applications can also be trivially lifted to an SMR-like semantics in case the developer does not need the additional detail provided (Section 3.2).

Like a standard sequential or concurrent object, an ADO has private state that can be atomically accessed and updated through a public interface of user-defined methods. There are, however, fundamental differences between concurrent and distributed objects [Waldo et al. 1994], which the ADO model reflects. For instance, due to a combination of replicated state and an unreliable network, methods may nondeterministically fail to reach certain servers, temporarily creating inconsistent states. Even if an application hides these from clients, how partial failures are handled heavily influences the consistent state that is eventually reached. Therefore, to offer application

Table 1. Comparison between distributed system verification frameworks.
* Only one or the other at a time (specifications use either SMR or network model).
† Ironclad [Hawblitzel et al. 2014] demonstrates how to translate Dafny into the BoogieX86 verifiable assembly.

	Atomic Interface	Exposes Failures	Horizontal Composition	Vertical Composition	Verified Executable	
ADVERT	✓	✓	✓	✓	C	✓
IronFleet	✓*	✓*	×	✓	C#	×†
Verdi	✓*	✓*	×	✓	OCaml	×
Disel	×	✓	✓	×	OCaml	×
Aneris	×	✓	✓	✓	None	×

designers precise control over intermediate states, the ADO interface also includes `push` and `pull` operations inspired by the push/pull shared memory model [Gu et al. 2016, 2018].

To demonstrate the ADO model's utility for practical distributed system development and verification, we also present ADVERT (atomic distributed object verification toolchain), a verification framework in the Coq proof assistant [The Coq Development Team 2018]. ADVERT consists of an implementation of the ADO model plus a collection of modular libraries of definitions and proofs targeted at different levels of end-to-end distributed system verification: constructing and reasoning about individual ADOs (Sections 2 and 3), composing ADOs (Section 4), and verifying executable implementations against ADO specifications (Section 5).

Table 1 summarizes the features supported by ADVERT compared to relevant previous work. We defer a more detailed comparison to Section 7, but the key difference is that ADVERT is the only framework to provide an atomic interface that also exposes partial failures. By an atomic interface we mean one in which client-application communication appears to happen instantaneously. ADVERT supports this through the ADO model's atomic `pull`, `push`, and method call operations. IronFleet and Verdi offer a choice between using an atomic SMR interface (implemented by multi-Paxos and Raft respectively), or writing applications directly in terms of network events, which are not atomic because even a simple message between servers consists of separate send and receive events. One can prove that these network events behave atomically, but it is not automatically guaranteed by the interface. Disel and Aneris also provide non-atomic, network-based semantics for application building. Nevertheless, we believe the ADO model and elements of ADVERT are compatible with previous work and should be seen as complementary tools rather than strict replacements.

Another area where ADVERT and the ADO model improve on existing work is support for compositional reasoning. This is an important topic because an incorrect interface between even two verified components can introduce serious bugs that threaten the whole system [Fonseca et al. 2017]. IronFleet [Hawblitzel et al. 2015a] and Verdi [Wilcox et al. 2015] support some form of vertical composition (decomposing complex proofs into simpler layers), Disel [Sergey et al. 2017] supports horizontal composition (decomposing large systems into independent components), and Aneris [Krogh-Jespersen et al. 2020] supports both. ADVERT also supports both forms of composition, and with its atomic interface one can more easily reason about complex composite applications, including ones without centralized coordinators ("lock-free" applications).

Additionally, while the other systems offer only unverified extraction to executable code, ADVERT supports end-to-end verification from an ADO specification down to an executable C implementation using certified concurrent abstraction layers (CCAL) [Gu et al. 2018].

This paper makes following contributions:

- A novel atomic object model that facilitates the design and verification of efficient and correct distributed applications. It provides a clear and precise semantics for strongly consistent protocols

that is generic and simple to use because it hides many implementation details. At the same time, it offers fine-grained control over inconsistent states by exposing the nondeterminism and important failure cases that arise from asynchronous networks.
- Advert, an end-to-end verification framework built around the ADO model in Coq.
- A case study comparing different distributed key-value store designs that support partitioning and replication via ADO composition. This includes the first, to our knowledge, machine-checked correctness proof of a distributed system composition without a centralized coordinator.
- A case study of Two-Phase Commit with replicated resource managers that demonstrates how the additional details exposed by the ADO model enable reasoning about optimizations that cannot be expressed in an SMR-like model.
- Refinement proofs between the ADO model and network-based specifications of several distributed protocols including multi-Paxos, Vertical Paxos, CASPaxos, and Chain Replication. This is the first machine-checked proof that these protocols have equivalent high-level behaviors. The multi-Paxos specification is additionally formally linked to a C implementation and an executable binary is generated by a verified compiler [Gu et al. 2015; Leroy 2009].

All of our Coq and C code is available at https://zenodo.org/record/5476274.

2 ATOMIC DISTRIBUTED OBJECT MODEL

The purpose of the ADO model is to offer a simple abstraction for modular reasoning about distributed objects. Before explaining how it accomplishes this goal, we first motivate the need for such a model by discussing the challenges of distributed systems and the limitations of existing models. We then present a high-level summary of the ADO model, demonstrate its relation to concrete protocols, and finally present the formal details.

2.1 Background and Motivation

Strong consistency is difficult to guarantee in a distributed setting in which physically isolated servers coordinate over an asynchronous and potentially faulty network. This, unsurprisingly, means that distributed protocols are complex and can exhibit unintuitive behaviors. Some of these oddities are implementation artifacts and should be abstracted away, but others represent core distributed features that cannot be ignored. This section begins with a brief primer on Paxos [Lamport 2001], a popular distributed protocol, and then highlights two important challenges in reasoning about it that the ADO model is designed to address: handling failures and composing systems.

Paxos. The high-level goal of Paxos is to replicate some state across a set of servers (or *replicas*). In order to provide a consistent view of the replicated state to clients, the protocol must achieve *consensus* among the replicas by getting a large-enough subset (typically a majority), called a *quorum*, to agree on the same value. The primary safety property guaranteed by Paxos is that once consensus is reached, the state is *committed* and immutable. Furthermore, this is ensured even if a subset of the replicas become unresponsive, as long as a quorum continues working.

Paxos works in rounds, each of which is identified by a unique *ballot number* (a kind of logical timestamp) and has a designated server called the *proposer* that suggests values for the replicas to commit. Each round consists of two phases called either phase 1 and phase 2 or `prepare` and `write`.

A proposer begins a `prepare` phase by suggesting a new ballot number to the replicas. A replica either responds positively if the ballot number is the largest it has seen thus far, or negatively otherwise. Positive responses include the replica's current state (or $\perp$ if no value has been proposed yet) along with the ballot number of the round in which it was proposed. A `prepare` request succeeds if it receives a quorum of positive acknowledgements, at which point the proposer selects the value with the largest ballot number. If it is $\perp$ the proposer is free to choose an arbitrary value.

Next, the proposer tries to `write` this value by again broadcasting to the replicas. Replicas confirm that the proposer's ballot number is still the most up-to-date, and if so they update their own state with the proposed value. If a quorum of replicas accept the value then it is committed, but if even one accepts it then it is *partially committed*. This is important, because another proposer might observe this value during a later `prepare` and finish committing it during its `write` phase.

Paxos is only able to reach consensus on a single value, but protocols like multi-Paxos [Renesse and Altinbuken 2015] extend it to safely replicate a sequence of values. The core principles are the same, but now each replica maintains a log of values, and the proposer's goal is to append new values to the end of the log while ensuring that a quorum agrees on all of the earlier entries. This is often used to implement a distributed state machine by replicating a log of *commands*, which are application-specific functions that can be applied in order to compute the current state (e.g., the log `add(3)` • `sub(1)` • `mul(3)` evaluates to 6 assuming an initial state of 0).

Failures. 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 and, as we have seen, can influence later committed states. The state machine replication (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 is achieved by a black-box remote procedure call (RPC) interface that is typically implemented by calling `prepare` followed by `write` and retrying each step until it succeeds.

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 3.2 describes some common method-calling patterns for various application requirements and shows how each is supported in the ADO model.

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 `prepare` phase and skipping the `write` 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. Similarly, in Raft a leader may continue accepting new log entries before the previous entries are fully committed. A client only sees the fully committed entries, but this optimization allows the leader to handle requests without blocking and to batch-process outstanding requests. Another interesting case is consensus combined with distributed transactions where partially committed states can be used as hints to speed up transaction decisions. We show this example in Section 4.2 and discuss in Section 7 how it relates to other systems with similar optimizations.

Composition. Protocols like multi-Paxos can implement a distributed object, but practical applications typically consist of many interacting objects. Furthermore, objects have varying performance requirements so ideally a distributed application should be implemented by a heterogeneous collection of protocols. It is not difficult to see that without a compositional abstraction, such a system would be hopelessly complex to verify. There are two axes we consider when discussing composition: vertical and horizontal. Vertical composition refers to relating a specification to a more abstract version of itself and transitively linking a series of these relations. This is necessary for hiding implementation details and is well supported by existing distributed verification frameworks such as Verdi [Wilcox et al. 2015] and IronFleet [Hawblitzel et al. 2015a].

Horizontal composition involves plugging two independently verified systems together to create a bigger system. This simplifies reasoning about large systems by allowing them to be broken into smaller components. It also provides an additional level of modularity as a component can be verified

once and reused by multiple applications. Neither Verdi nor IronFleet support communication between components in separate systems, so horizontal composition is impossible. Disel [Sergey et al. 2017] and Aneris [Krogh-Jespersen et al. 2020] on the other hand are designed for this type of interaction; however, it comes at the cost of a lower level of abstraction that is difficult to scale to larger examples. See Section 7 for further discussion about these frameworks. Section 4 provides several examples of both vertical and horizontal composition with the ADO model.

2.2 Modifying the Push/Pull Model

While designing a distributed object model that can handle these challenges, we took inspiration from shared memory concurrent object models. In particular, we found that the push/pull shared memory model [Gu et al. 2016, 2018] accomplishes our goals of compositionality and hiding complexity and has an interface that maps nicely onto the two-phase design of many distributed protocol. 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 simple FIFO `Queue` in Fig. 1 as a running example of an ADO. Section 3 explains the notation further, but for now it is sufficient to understand that this represents a queue with atomic methods that is implemented on a distributed `Vector` (resizable array).

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

Fig. 1. FIFO Queue object.

Push/Pull Basics. A core element of an object model is its state representation. The push/pull model represents an object's state as a logical history of the methods called up to that point (e.g., enqueue(1) • 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 purely a logical tool used at the specification level. Even though it resembles the logs used in state machine replication, implementations can still perform in-place updates instead (e.g., as in CASPaxos [Rystsov 2018]).

The other key component of an object model is the interface through which clients interact with the state. In the push/pull model this consists of methods (e.g., `enqueue`), and two special operations for managing concurrent access: `pull` and `push`. Before applying a method 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 and during this time other clients cannot access the object. The client then applies methods to the local copy and calls `push` to copy back the updates and release ownership. Note that methods are asynchronous because the application and return points (`push`) are separate events. To distinguish between the single event of a local method application and the entire sequence of `pull`, method, and `push`, we refer to the former as a *method invocation* and the latter as a *method call*.

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, it is an unreasonably strong restriction to completely block other clients when one owns the object. Due to an unreliable network, a client may become unresponsive, which could prevent others from making progress. Similarly, random network and node failures can significantly delay messages or prevent them from being delivered, so updates are not guaranteed to succeed. Finally, distributed protocols replicate state across multiple servers, some subset of which must agree for a state to be considered consistent. This allows for partial failures; i.e., situations in which an attempted update reaches an insufficient number of servers to be consistent, but still enough to influence subsequent updates.

Table 2. Mapping ADO operations to distributed protocols.

ADO	Multi-Paxos	Chain Replication	Raft
Pull	Phase 1	Read from tail	Election
Method	Local update	Write to head	Local update
Push	Phase 2	Send down chain	Log replication

Preemptible Ownership and Nondeterminism. The first problem is solved by making ownership preemptible. Clients still copy the state and mark their ownership with `pull`, but this no longer locks the state. If a client maintains ownership then `push` behaves as before, but it fails if another client preempts it with `pull` first. To model network and node failures `pull` and `push` are also allowed to fail nondeterministically at any time.

Partial Failures. Handling the intermediate states introduced by partial failures requires the most significant change. The state is extended to include both the log of consistent updates, and a tree of proposed updates that failed to achieve consensus. We refer to the log as the *persistent state* because its entries never change after they are added. Continuing the shared memory analogy, we call the elements of the tree *volatile caches* because their effects are visible, but temporary unless they are flushed to the persistent state. New updates must build off of either a volatile cache or the latest entry in the persistent state. This dependency is represented through the parent-child relation in the cache tree with the persistent state at the root.

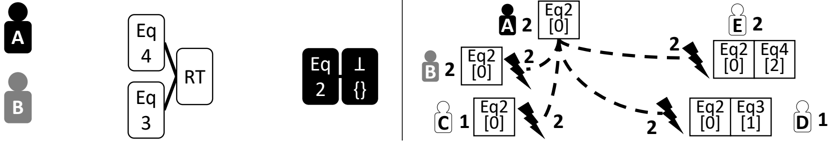
Instead of creating a local copy of the persistent state, `pull` now marks an arbitrary cache in the tree (or the root) as "active". This reflects the fact that clients may observe different snapshots of the state depending on which servers are contacted. Invoking a method creates a new branch rooted at this cache and updates the active position. A successful `push` moves the entire active branch to the persistent state. All other branches are removed because they represent unreachable states that no longer depend on the new persistent state.

Multiple Methods per `push`. A client may invoke multiple methods in the "critical section" between `pull` and `push`. This models situations as in Raft where a leader can accumulate several updates in its local log before beginning to replicate them. If the pending updates are sent in a batch then they will all succeed or fail together, but to remain as general as possible, we assume each is delivered by a separate message, and therefore one or more might fail independently from the others. To avoid gaps in the log, when one update fails all of the following ones must be rejected as well. Therefore, when `push` moves the active branch into the persistent state it now leaves an arbitrary (possibly empty) suffix of failed caches.

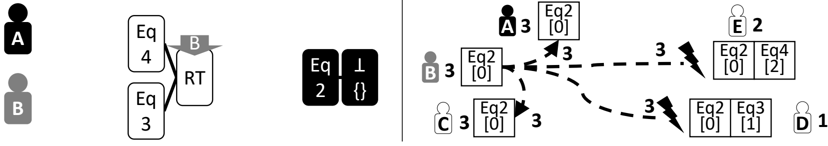
2.3 Connection with Distributed Protocols

ADOs capture the common high-level behavior of strongly consistent systems, such as probing existing states, handling intermediate failures, and clients competing to commit new states. Therefore, by design, there is a close correspondence between ADO operations and those in protocols such as Paxos and Raft (see Table 2). Fig. 2 visualizes this mapping with a multi-Paxos implementation of the `Queue` object from Fig. 1 alongside its corresponding ADO representation. We then step through a sequence of events (labeled **a**-**f**) to show how each model evolves over time.

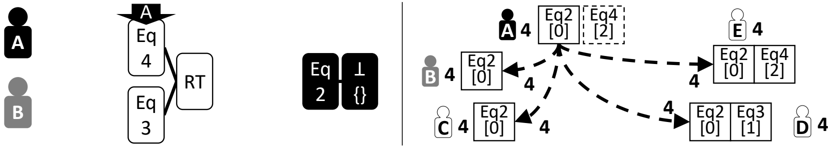
On the multi-Paxos side, there are five replicas (**A**-**E**), two of which (**A** and **B**) are also proposers. Each replica has a ballot number and a local log. The log entries contain a method (e.g., `Eqx`, which stands for `enqueue(x)`) as well as the ballot number for the round in which it was added to the log. Messages between replicas are represented as arrows (dotted for phase 1, solid for phase 2). A lightning bolt indicates that a replica did not receive the message due to a network error.



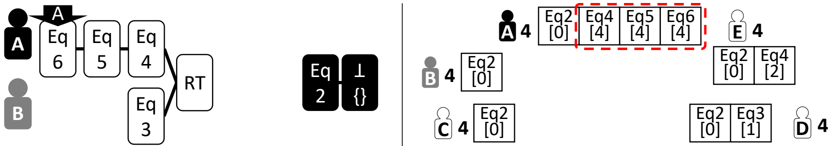
(a) ADO pull and multi-Paxos phase 1 failure due to a network disconnection.



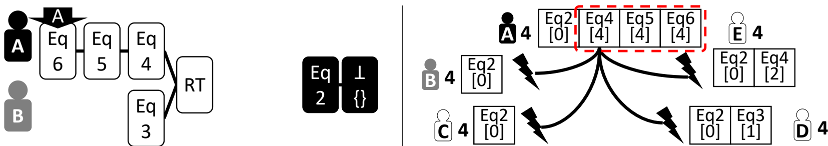
(b) ADO pull and multi-Paxos phase 1 success (**B** succeeds with **A** and **C**'s support).



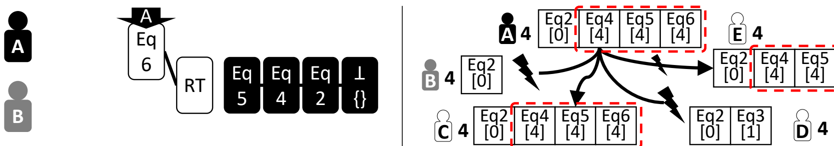
(c) ADO pull and multi-Paxos phase 1 preemption (**A** proposes a larger ballot number than **B** and finds an uncommitted log entry Eq4; Eq3 is ignored because its ballot number is smaller than that of Eq4).



(d) ADO method invocation and multi-Paxos local log update (within **A**, Eq4's ballot number is updated and Eq5 and Eq6 are added locally).



(e) ADO push and multi-Paxos phase 2 failure due to a network disconnection.



(f) ADO push and multi-Paxos phase 2 partial success due to a network failure (Eq4 and Eq5 are successfully committed to a quorum consisting of **A**, **C** and **E**, but Eq6 is not fully committed).

Fig. 2. ADO (left) and multi-Paxos (right) mapping. Multi-Paxos requires a majority support for successful transitions. The ADO's cache tree and persistent log abstract away individual replicas. The figures show a sequence of continuous operations.

The ADO side of the diagram shows the cache tree (the white boxes), which begins at the root (marked RT) and grows to the left, as well as the persistent log (the black boxes), which also grows to the left. Each box contains a method. The current owner's active cache is marked with an arrow.

(a) Pull failure: We begin in a state where `Eq2` is the only committed method (it is in every replica's local log and the ADO's persistent log). Replicas **D** and **E** each also have one uncommitted method (`Eq3` and `Eq4`, respectively), which manifests as two entries in the cache tree. Proposer **A** attempts to become the owner by broadcasting a phase 1 request, but fails because its messages are dropped. In the ADO model this corresponds to `pull` failing nondeterministically. In this case neither the replicas' logs nor the cache tree change.

(b) Pull success: Proposer **B** now attempts to become the owner and succeeds by demonstrating to a quorum of voters (**A** and **C**) that it has the largest ballot number. The voters update their ballot numbers and respond to **B** with their current logs, who then selects the one with the largest ballot number in the last entry. In this case both voters have the same log, which contains only the committed entry `Eq2`. The corresponding result in the ADO model is for `pull` to succeed and set **B**'s active cache to the root of the cache tree.

(c) Pull preemption: Proposer **A** then tries again to become the owner, and this time it succeeds with every replica's vote, taking ownership away from **B**. Of the voters, **E**'s log is the most up-to-date, so Proposer **A** copies it. The ADO represents **A**'s uncommitted log entry by setting its active cache to the one containing `Eq4`.

(d) Method invocation: As the owner, **A** can now call new methods. It chooses to enqueue 5 and 6, but before it replicates these methods to the other replicas it first adds them to its own log. In the cache tree this is represented by extending **A**'s active branch with two new caches.

(e) Push failure: Next, **A** tries to commit the new methods, but before it can commit `Eq6` it must first commit `Eq4` and `Eq5` in that order. However, upon broadcasting a phase 2 request to commit `Eq4`, the network drops its messages so the commit fails. The ADO model captures this situation by having `push` fail. As in the previous failure case the state is unchanged.

(f) Push success: Noticing the failure, **A** retries and this time manages to reach **C** and **E**. As this constitutes a quorum (3 out of 5), `Eq4` is successfully committed so it moves on to `Eq5`, which also succeeds. Finally it attempts to commit `Eq6`, but only **C** receives the message so it is not committed. The result is that there are three committed methods (`Eq2`, `Eq4`, `Eq5`), and two uncommitted methods (`Eq3` and `Eq6`). However, note that `Eq3` in **D**'s log has a ballot number of 1. This is smaller than the ballot number for `Eq5` and `Eq6` (4), so there is no quorum in which **D**'s log is the most up-to-date. Therefore, this method is unreachable so it is removed from the cache tree. `Eq6` on the other hand also has a ballot number of 4, and is still a viable option for `pull` to choose as an active cache, so it remains in the tree.

2.4 ADO Formal Semantics

We now formalize the extended push/pull model described in the previous sections. An ADO (Fig. 3) is a pair of the object's internal state of type Σ (e.g., `Vector[ℤ]` in `Queue`), and a method interface. A method interface is a partial map from *Methods* to method bodies; i.e., functions from Σ to Σ plus a return value R. In practice, methods and their arguments are serialized and packed together into a network packet, so for simplicity we assume the parameters are encoded in the method name. For example, `Queue`'s method interface includes `dequeue()`, `enqueue(1)`, `enqueue(2)`, and so on.

The global system state is called a Distributed Shared Memory (*DSM*), which consists of a list of persistent methods (*PersistLog*), a tree of volatile caches containing not-yet committed methods (*CacheTree*), a partial map that remembers each client's active position in the cache tree (*CIDMap*), and another partial map from a logical timestamp (e.g., a Paxos ballot number or Raft term number)

$$
\begin{aligned}
ADO^{\Sigma} &\triangleq \Sigma * (\exists R.\, Method \rightharpoonup (\Sigma \to \Sigma * R)) \\
CID &\triangleq \langle \mathbb{N}_{nid} * \mathbb{N}_{time} * CID \rangle \mid \mathbf{Root} \\
Cache &\triangleq CID * Method \\
PersistLog &\triangleq List(Cache) \\
CacheTree &\triangleq Set(Cache) \\
CIDMap &\triangleq \mathbb{N}_{nid} \rightharpoonup CID \\
OwnerMap &\triangleq \mathbb{N}_{time} \rightharpoonup (\mathbb{N}_{nid} \mid \mathbf{NoOwn}) \\
DSM &\triangleq PersistLog * CacheTree * CIDMap * OwnerMap
\end{aligned}
$$

$$
\begin{aligned}
ADOEvent \triangleq\ & Pull^{+}(\mathbb{N}_{nid} * \mathbb{N}_{time} * CID) \\
& \mid Pull^{*}(\mathbb{N}_{nid} * \mathbb{N}_{time}) \\
& \mid Pull^{-}(\mathbb{N}_{nid}) \\
& \mid Invoke^{+}(\mathbb{N}_{nid} * Method) \\
& \mid Invoke^{-}(\mathbb{N}_{nid}) \\
& \mid Push^{+}(\mathbb{N}_{nid} * CID) \\
& \mid Push^{-}(\mathbb{N}_{nid}) \\
ADOLog \triangleq\ & List(ADOEvent)
\end{aligned}
$$

$$
\begin{aligned}
nidOf(cid) &\triangleq \mathbf{let}\ \langle nid, _, _ \rangle = cid\ \mathbf{in}\ nid \\
timeOf(cid) &\triangleq \mathbf{let}\ \langle _, time, _ \rangle = cid\ \mathbf{in}\ time \\
nextCID(cid) &\triangleq \langle nidOf(cid), timeOf(cid), cid \rangle
\end{aligned}
$$

$$
\begin{aligned}
cid_1 < cid_2 &\triangleq cid_2 \neq \mathbf{Root} \land \mathbf{let}\ \langle _, _, parent \rangle = cid_2 \\
&\quad\ \mathbf{in}\ cid_1 = parent \lor cid_1 < parent \\
cid_1 \leq cid_2 &\triangleq cid_1 < cid_2 \lor cid_1 = cid_2
\end{aligned}
$$

Fig. 3. ADO state definitions and *CID* helper functions.

GenPullSuccess

$$
\frac{\mathbb{O}_{pull}(log, nid) = Ok(time, cid) \qquad timeOf(cid) < time \qquad noOwnerAt(log, time) \qquad (cid \in caches(log) \lor cid = root(log))}{\mathbb{O}_{pull} \vdash pull(nid) : log \longrightarrow log \bullet Pull^{+}(nid, time, cid)}
$$

GenPullPreempt

$$
\frac{\mathbb{O}_{pull}(log, nid) = Preempt(time) \qquad time \notin dom(owners(log))}{\mathbb{O}_{pull} \vdash pull(nid) : log \longrightarrow log \bullet Pull^{*}(nid, time)}
$$

GenMethodInvocation

$$
\frac{cids(log)[nid] \in caches(log)}{\vdash M(nid) : log \longrightarrow log \bullet Invoke^{+}(nid, M)}
$$

GenPushSuccess

$$
\frac{\mathbb{O}_{push}(log, nid) = Ok(ccid) \qquad nidOf(ccid) = maxOwner(log) = nid \qquad timeOf(ccid) = timeOf(cids(log)[nid]) \qquad ccid \in caches(log)}{\mathbb{O}_{push} \vdash push(nid) : log \longrightarrow log \bullet Push^{+}(nid, ccid)}
$$

$$
\begin{aligned}
\mathbb{O}_{pull} &: ADOLog \to \mathbb{N}_{nid} \to (Ok(\mathbb{N}_{time} * CID) \mid Preempt(\mathbb{N}_{time}) \mid Fail) \\
\mathbb{O}_{push} &: ADOLog \to \mathbb{N}_{nid} \to (Ok(CID) \mid Fail) \\
root(log) &\triangleq \mathbf{let}\ (p, _, _, _) = interpADO(log)\ \mathbf{in\ if}\ p \neq []\ \mathbf{then}\ last(p)\ \mathbf{else\ Root} \\
caches(log) &\triangleq \mathbf{let}\ (_, cs, _, _) = interpADO(log)\ \mathbf{in}\ cs \\
cids(log) &\triangleq \mathbf{let}\ (_, _, cids, _) = interpADO(log)\ \mathbf{in}\ cids \\
owners(log) &\triangleq \mathbf{let}\ (_, _, _, owns) = interpADO(log)\ \mathbf{in}\ owns \\
noOwnerAt(log, time) &\triangleq time \notin dom(owners(log)) \lor owners(log)[time] = \mathbf{NoOwn} \\
maxOwner(log) &\triangleq \mathbf{let}\ owns = owners(log)\ \mathbf{in}\ owns[max(dom(owns))]
\end{aligned}
$$

Fig. 4. Selected ADO log generation rules and utility functions.

to its unique owner (*OwnerMap*). A *Cache* is a method paired with a unique cache ID (*CID*), which is inductively defined as either **Root**, or a triple of a node ID ($\mathbb{N}_{nid}$), a logical timestamp ($\mathbb{N}_{time}$), and a parent *CID*. Intuitively, a *CID* represents a branch in the cache tree (note that it is isomorphic to $List(\mathbb{N}_{nid} * \mathbb{N}_{time})$), and the tree itself is simply a prefix-closed set of *Caches*. In general, *CIDs* are only partially ordered by their parent-child relationship, but the persistent log maintains a totally ordered, sorted list, which guarantees that methods are applied in the order they were called.

Log Generation. An ADO's interface consists of its methods, `pull`, and `push`. Results of these operations are recorded in a log of *ADOEvents*. Fig. 4 defines each operation's effect on the current log.[1] Clients trigger these events by executing sequences of ADO operations. Multiple clients may concurrently access an ADO so the events generated by each client may interleave in the log (see Section 4 for examples). The details of how a client communicates with an object (e.g., discovering the leader) are left up to the implementation and are not exposed at the ADO level.

Calling `pull` results in a $Pull^+$ event on success, $Pull^-$ on failure, or $Pull^*$ in the event that the client failed to become an owner itself, but had enough support to strip another client's ownership. 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* ($\mathbb{O}_{pull}$). This is an abstract function that, given the current log and the caller's unique ID, returns a new timestamp and an arbitrary position in the cache tree (just the timestamp in the preemption case), or else indicates that `pull` failed completely. Oracles are deterministic, but network-based nondeterminism is modeled by quantifying over all valid oracles. In order to faithfully model protocols like Paxos, valid oracles must satisfy the side conditions in GenPullSuccess and GenPullPreempt; i.e., the new time is strictly larger than that of the chosen cache, the cache exists in the tree, and there is not already an owner at that time. The special value **NoOwn** indicates that a previous `pull` at some time t failed to achieve ownership, but preempted owners at times less than t.

Invoking a method adds an $Invoke^+$ event via GenMethodInvocation if the caller's active *CID* is still in the cache tree. This also implies that the caller first became an owner since $Pull^+$ is the only way to add a new mapping to the *CIDMap*. This check also ensures that clients cannot continue working on a branch that has been invalidated by a successful `push`.

As with `pull`, `push` succeeds ($Push^+$) or fails ($Push^-$) based on the outcome of $\mathbb{O}_{push}$. Recall that when a client calls `push` after a sequence of method invocations, some suffix may fail. $\mathbb{O}_{push}$ captures this in GenPushSuccess by choosing an arbitrary cache, *ccid*, from the active branch and committing everything up to that point. One important restriction is that the caller must be not only an owner, but the owner currently with the largest timestamp. This is necessary in protocols like multi-Paxos and Raft to ensure that only the most up-to-date state is committed.

Log Interpretation. Given an *ADOLog*, it can be interpreted by the *interpADO* function (Fig. 5) to construct a *DSM*. PullSuccess simply updates the *CIDMap* and *OwnerMap* with the caller's information. PullPreempt does the same but only for the *OwnerMap*. The function *voteNoOwn* takes a time and fills in all empty slots in the *OwnerMap* below that time with **NoOwn**. This is because in multi-Paxos or Raft, once a server has seen a message with timestamp t, it will reject all future election or commit attempts with timestamps less than or equal to t.

MethodInvocation adds a new entry to the caller's active branch in the cache tree and then updates its position in the *CIDMap*. PushSuccess partitions the tree around *ccid* into successful and failed caches. The successful caches are the ancestors of *ccid*, which are appended to the persistent state. The failed caches are the sibling branches, which lack the dependency on the newly persistent states and are thus removed from the tree. The descendants of *ccid* (the failed suffix) do maintain this dependency, so they remain in the tree and can potentially be committed by a later `push`. The failure events ($Pull^-$, $Invoke^-$, and $Push^-$) represent cases where the caller had no effect on the global state so *interpADO* treats them as no-ops (omitted here for space).

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

[1] Some failure cases are omitted for space. See Appendix A for the complete semantics.

$$\textsc{PullSuccess}\;\frac{cids' = cids[nid \mapsto \langle nid, time, cid\rangle] \qquad owns' = voteNoOwn(owns[time \mapsto nid], time-1)}{Pull^{+}(nid, time, cid) : (p, cs, cids, owns) \longrightarrow (p, cs, cids', owns')}$$

$$\textsc{PullPreempt}\;\frac{owns' = voteNoOwn(owns, time)}{Pull^{*}(nid, time) : (p, cs, cids, owns) \longrightarrow (p, cs, cids, owns')}$$

$$\textsc{MethodInvocation}\;\frac{cs' = cs \cup \{(cids[nid], M)\} \qquad cids' = cids[nid \mapsto nextCID(cid)]}{Invoke^{+}(nid, M) : (p, cs, cids, owns) \longrightarrow (p, cs', cids', owns)}$$

$$\textsc{PushSuccess}\;\frac{(\vec{c}_{ok}, cs') = partition(cs, ccid)}{Push^{+}(nid, ccid) : (p, cs, cids, owns) \longrightarrow (p \bullet \vec{c}_{ok}, cs', cid, owns)}$$

$$voteNoOwn(owns, time) \triangleq owns[t \mapsto \textbf{NoOwn} \mid \forall t \leq time.\, t \notin dom(owns)]$$

$$\begin{aligned} partition(cs, cid) \triangleq\ & \textbf{let } \vec{c}_{ok} = sort(\{(c, M) \in cs \mid c \leq cid\}) \textbf{ in} \\ & \textbf{let } cs' = \{(c, M) \in cs \mid cid < c\} \textbf{ in } (\vec{c}_{ok}, cs') \end{aligned}$$

Fig. 5. Selected ADO log interpretation rules (*interpADO*).

```
1 ADO BankAccount {
2   shared balance : ℤ := 0; /* Σ = ℤ */
3   /* read() ↦ λ bal. (bal, bal) */
4   method read() { return this.balance; }
5   /* deposit(n) ↦ λ bal. (bal + n, tt) */
6   method deposit(n) { this.balance += n; }
8   /* withdraw(n) ↦ λ bal.
9   *   if n ≤ bal then (bal - n, n) else (bal, 0) */
10  method withdraw(n) {
11    if (n ≤ this.balance) {
12      this.balance -= n; return n; }
13    else { return 0; } } }
```

Fig. 6. Distributed bank account object.

example ADOs use an object-oriented pseudocode rather than the formal log-based representation from Fig. 3. For example, in the simple `BankAccount` ADO (Fig. 6), 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). A client with node ID `nid` interacts with an object `obj` by calling `obj.pull<nid>()`, `obj.push<nid>()`, or `obj.m<nid>()` for some method `m` (`<nid>` is omitted when it is clear from context). The caller's node ID can be accessed within a method body with `this.nid`.

3.1 ADO Properties

One property the ADO model guarantees is, unsurprisingly, atomicity. Intuitively, an operation is atomic if it executes in a single instant with no opportunity to interleave with other operations. More precisely an ADO operation is atomic if it generates exactly one ADO event.

```
1 Definition atomic (f: ADOLog -> ADOLog) := forall log, exists ev, f log = log ++ [ev].
```

From Fig. 4 it is clear that every ADO operation (`pull`, `push`, and method invocation) is atomic. This means it is impossible, for example, for a method invocation to add an entry to the cache tree at the same time as `push` prunes invalid branches. This atomic semantics is one of the main strengths of the ADO model compared to network-based models. For example, a Paxos `prepare`

request broadcasts a message to the acceptor nodes and collects their replies. In terms of network events (sends and receives), this is clearly non-atomic as there are many permutations in which concurrent operations can interleave. Although one can prove that Paxos ensures that clients observe state updates as if they happened atomically, the ADO model makes this guarantee explicit by refining the non-atomic sequences of network events into atomic ADO events (see Section 5.2).

Another property guaranteed by the ADO semantics is *Replicated State Safety*; i.e., the persistent log at some time is a prefix of the log at a later time, which one can easily verify by noting that all of the cases in *interpADO* only ever append to the persistent log. Because the client-observable state is computed from the persistent log, this property also implies strong consistency; i.e., all clients observe state updates in the same order.

```
1 Theorem repStateSafety : forall (evs evs': ADOLog),
2   let log := (interpADO evs).(persist) in let log' := (interpADO (evs ++ evs')).(persist) in
3   log = firstn (length log) log'. (* firstn n xs returns the first min(n, length xs) elements of xs *)
```

3.2 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 can interleave with concurrent calls. 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 remote procedure call (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. In the ADO model, this means there is no more than one cache per node ID with the at-most-once-called method in the persistent log.

```
1 Definition called (log: ADOLog) (nid: nat) (m: Method) (c: Cache) :=
2   c.(nid) = nid /\ c.(method) = m /\ In c (interpADO log).(persist).
3 Definition at_most_once (log: ADOLog) (nid: nat) (m: Method) :=
4   (exists! c, called log nid m c) \/ (forall c, ~called log nid m c).
```

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). A method can be called with at-most-once semantics by calling `pull`, invoking the method, then calling `push`, and aborting if any step fails. We abbreviate this sequence of operations as `obj.m()?`. In the following syntax `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`**. Recall that, formally, these are functions on an ADOLog, but for simplicity we use this more concise imperative notation in which the log is implicitly threaded through each operation.

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

Note that `at_most_once` does not allow `nid` to ever call the same method twice. 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 for new calls.

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 even a failed method might be added to the cache tree and committed by a later `push`. Thus, retrying a method until it succeeds only guarantees that it is called *at least* once.

```
Definition at_least_once (log: ADOLog) (nid: nat) (m: Method) := exists c, called log nid m c.
```

This is appropriate for read-only methods (e.g., `read`), or if one only cares about the return value rather than the object's internal state (e.g., a random number generator's seed). One can make an at-least-once call (`obj.m()`$^{+}$) by repeating `obj.m()?` with a new request ID each time.[2]

```
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, as with at-least-once calls, the potential for partial failures means one cannot guarantee a method is not committed more than once; however, for idempotent methods, `obj.m()!` can achieve an equivalent result by repeating `obj.m()?` with the same request ID.

```
Definition exactly_once (log: ADOLog) (nid: nat) (m: Method) :=
  exists c, called log nid m c /\ interpADO log = interpADO (removeDups c log).
```

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

A method can be mechanically transformed into an idempotent one by simply caching the result using the node and request IDs as a key. This ensures that the method's side effects only execute once, and subsequent calls return the memoized value. For example, `withdraw` becomes:

```
method withdraw(rqID, n) {
  if ((this.nid, rqID) ∈ this.cache) { }
  else if (n ≤ this.balance) { this.balance -= n; this.cache[(this.nid, rqID)] := n; }
  else { this.cache[(this.nid, rqID)] := 0; }
  return this.cache[(this.nid, rqID)]; }
```

3.3 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 (`replay_st` computes the internal state from the persistent log).

```
Definition replay_st {T} {ado: ADO T} (persist: list (@Cache ado)) : T :=
  fold_left (fun st m => fst (m st)) persist ado.(default).
Lemma balance_nonneg : forall (log: ADOLog), 0 <= replay_st (interpADO log).(persist).
```

Proof. Proceed by induction on `log`. The base case is trivial since the initial balance is `0`. In the inductive case a new ADOEvent is appended to the log. For every event but $Push^{+}$ the persistent state does not change and the balance is non-negative by the inductive hypothesis. For $Push^{+}$, consider each of the methods that could be added to the persistent log: `read` is read-only; `deposit` only increases the balance; and `withdraw` ensures that the amount to be deducted is not greater than the balance. Therefore, the invariant holds for every `log` and the balance is never negative. □

`BankAccount` in Fig. 6 is only a specification and must be implemented by a distributed 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.

[2]Without liveness assumptions, this actually guarantees at-least-once behavior or an infinite loop. For simplicity, we assume the loop terminates and leave liveness considerations as future work.

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

Fig. 7. Single ADO key-value store.

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 in the ADO model than in 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 3.2).

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` (Fig. 6) to allow transfers between them could look like the following.

```
1 DApp TransferAccount(acct1: BankAccount, acct2: BankAccount) {
2   proc transfer12(n) { if (acct1.withdraw(n)! = n) { acct2.deposit(n)!; } }
3   proc transfer21(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 `transfer12` and `transfer21` 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.

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

The first example, `KVPrimitive` (Fig. 7), 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 and cannot go out of sync.

4.1.1 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 to reduce the risk of losing both to node failures. `KVLock` enables this type of implementation choice by composing separate `DVector` objects (an ADO wrapper around a sequential `Vector`) for the data and metadata (Fig. 8). These are then composed with a distributed lock (`CASLock`) for synchronization.

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

1 ADO CASLock {
2   shared owner : option ℕ := None;
3   method tryAcquire() {
4     if (this.owner = None) {
5       this.owner := Some(this.nid); }
6     return this.owner = Some(this.nid); }
7   method release() {
8     if (this.owner = Some(this.nid)) {
9       this.owner := None; } } }
```

```
1 DApp KVLock(
2     lk: CASLock, data: DVector[ℤ], meta: DVector[ℤ]) {
3   proc set(k, v) {
4     while (!this.lk.tryAcquire()!) {}
5     this.meta.insert(hash(k), sizeof(v))!;
6     this.data.insert(hash(k), v)!;
7     this.lk.release()!; }
8   proc lookup(k) {
9     while (!this.lk.tryAcquire()!) {}
10    v := this.data.get(hash(k))!;
11    this.lk.release()!;
12    return v; }
13  proc getmeta(k) {
14    while (!this.lk.tryAcquire()!) {}
15    m := this.meta.get(hash(k))!;
16    this.lk.release()!;
17    return m; } }
```

Fig. 8. Lock-based composite ADO key-value store.

For `KVLock` to implement `KVPrimitive`, its procedures must be atomic. 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 B.1).

```
1 Lemma mutex : forall (log log': @ADOLog CASLock) (nid: nat),
2   let persist := (interpADO log).(persist) in let persist' := (interpADO (log ++ log')).(persist) in
3   let owner := replay_st persist in let owner' := replay_st persist' in
4   owner = Some nid -> owner <> owner' ->
5   (* skipn n xs returns the remainder of xs after dropping the first n elements *)
6   exists (c: Cache), c.(nid) = nid /\ c.(method) = release(nid) /\ In c (skipn (length persist) persist').
```

Proof. Proceed by induction on `log'`. In the base case `owner = owner'`, which contradicts the hypothesis that `owner <> owner'`. In the inductive case, `persist'` and `owner'` are unchanged by all events but $Push^+$, so the property holds by the inductive hypothesis. In the $Push^+$ case some list of methods is appended to `persist'`. If it contains `release(nid)`, then we are done. Otherwise, the methods must either be `tryAcquire(*)`, or `release(nid')` for some `nid <> nid'`. Neither case changes `owner'`, so `owner <> owner'` still holds and the inductive hypothesis still applies. □

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 `set` since `lookup` and `getmeta` do not modify the key-value state and are simpler.

```
1 Definition R (persist_lk: list (@Cache KVLock)) (persist_prim: list (@Cache KVPrimitive)) :=
2   let kv := replay_st persist_prim in
3   let (lk, data, meta) := replay_st persist_lk in
4   forall k, kv @ hash k = (data @ hash k, meta @ hash k).
5 Lemma R_set : forall persist_lk persist_prim c_lk c_prim k v,
6   c_lk.(method) = KVLock.set(k, v) ->
7   c_prim.(method) = KVPrimitive.set(k, v) ->
8   c_lk.(nid) = c_prim.(nid) ->
9   R persist_lk persist_prim ->
10  R (persist_lk ++ [c_lk]) (persist_prim ++ [c_prim]).
```

```
1 DApp KVLockFree(data: DVector[ℤ], meta: DVector[ℤ∗ℤ]) {
2   proc set(k, v) {
3     idx := this.data.append(v)!;
4     this.meta.insert(hash(k), (sizeof(v), idx))!; }
5   proc lookup(k) {
6     (_, idx) := this.meta.get(hash(k))!;
7     return this.data.get(idx)!; }
8   proc getmeta(k) { (m, _) := this.meta.get(hash(k))!; return m; } }
```

Fig. 9. Lock-free composite key-value store.

Proof. Choose any key `k'` and let `kv` be the state of `KVPrimitive` before `set`, `kv'` be the state after, and similarly for `data` and `meta`. We must show `kv' @ hash(k') = (data' @ hash(k'), meta' @ hash(k'))`. If `hash(k) = hash(k')`, then `KVPrimitive.set(k, v)` means `kv' @ hash(k') = (sizeof(v), v)`. `KVLock` uses only exactly-once method calls, which we assume eventually succeed. After acquiring the lock, it inserts into `meta` and `data`. Now `meta' @ hash(k') = sizeof (v)` and `data' @ hash(k') = v`, so the relation holds. If `hash(k) <> hash(k')` then `set` does not change the mapping at `k` so `kv' @ hash(k') = kv @ hash(k')` and likewise for `meta` and `data`, so the relation holds by hypothesis. □

4.1.2 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` (Fig. 9) 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 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 can be read by a client). We sketch the case where `set` and `lookup` are executed concurrently with the same key (`k`) by threads `T1` and `T2` respectively. The other cases are similar. The two ways in which the procedures can interleave are:

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

or

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

Proof. At the beginning of each case we have the invariant that $\forall$ `idx ∈ map(snd, meta). data.len > idx`. This can be seen by observing that only `set` modifies `data` or `meta` and the index it inserts equals `data.len - 1`. 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 commute (Lines 2 and 3 in the left case). In the right case, however, the operations on `meta` are in between the `data` operations. Since `meta` and `data` are separate objects, their methods may also commute as long as program order is preserved (Lines 1 and 2 & Lines 3 and 4). Thus, after reordering, both cases are equivalent to atomically executing `lookup(k)` followed by `set(k, v)`. □

```
1 DECISION := YES | NO | COMMIT | ABORT;
2 TX := {ops: Vector[IO]; ts: ℤ; decision: DECISION};
3 /* Local decision to vote YES or NO if tx can be
4    applied to txs */
5 func tx_can_commit(txs, tx) { ... }
6 ADO RM {
7   shared txs : Vector[TX] := [];
8   method prepare(tx) {
9     tx.decision := tx_can_commit(this.txs, tx);
10    this.txs.append(tx);
11    return tx.decision; }
12  method decide(ts, decision) {
13    idx := this.txs.find(λ tx. tx.ts = ts);
14    this.txs[idx].decision := decision; } }
```

```
1 DApp TM(rm_1: RM, ..., rm_n: RM) {
2   local ts : ℤ := 0;
3   /* Must be called once when TM starts */
4   proc init() {
5     for rm in [this.rm_1, ..., this.rm_n] {
6       while (rm.pull() = FAIL) {} } }
7   proc handle_request(ops) {
8     this.ts += 1;
9     tx := {ops=ops, ts=this.ts, decision=COMMIT};
10    /* Phase 1: Collect decisions */
11    for rm in [this.rm_1, ..., this.rm_n] {
12      /* Method invocation only, not an
13         exactly-once call */
14      rm.prepare(tx);
15      for i in 0..MAX_TRY {
16        res := rm.push();
17        if (res != FAIL) { break; } }
18      /* Abort and break if RM says no or can't
19         commit in MAX_TRY tries */
20      if (res = NO || res = FAIL) {
21        tx.decision := ABORT;
22        break; } }
23    /* Phase 2: Commit the decision */
24    for rm in [this.rm_1, ..., this.rm_n] {
25      rm.decide(tx.ts, tx.decision);
26      while (rm.push() = FAIL) {} } } }
```

Fig. 10. 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)) { return; }
9     /* Phase 2: Commit the decision */
10    for txs in [this.txs_1, ..., this.txs_n] { txs.append(tx); } }
```

Fig. 11. Transaction ADO.

Lock-free systems can be much more performant than lock-based ones, but to our knowledge no other verification framework has verified an example like KVLockFree. However, with the ADO model, reasoning about it is comparable to working in a shared-memory concurrent setting.

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. In the ADO model, 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) that each store 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 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.

Fig. 10 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 requests one at a time. A more realistic version that properly handles state recovery after a TM crash can be found in Appendix B.2. The code is mostly unsurprising, but there are two points that deserve attention.

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 risk of preemption. This means that, unlike exactly-once calls, the method calls on Lines 14 and 25 can skip calling `pull`.

Note that `KVLock` also guarantees that only the client that holds the lock modifies `data` and `meta`. One could therefore use similar optimizations to improve the performance of a procedure that inserts a batch of key-value pairs by acquiring the lock, calling `pull` once, invoking `data.insert` and `meta.insert` for each key-value pair, then calling `push` until they all succeed.

The second place that Fig. 10 differs from earlier examples is Line 15 in `handle_request`. Unlike exactly-once calls, which retry `push` infinitely, this limits 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 the ADO model facilitates optimized system designs.

As in the previous examples we can show that this DApp refines an ADO specification (Fig. 11). 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.

```
1 Definition R (persist_2pc: list (@Cache TM)) (persist_tx: list (@Cache Transaction)) :=
2   let (ts, tx_1, ..., tx_n) := replay_st persist_tx in let (rm_1, ..., rm_n) := replay_st persist_2pc in
3   forall i tx, tx.(decision) = COMMIT -> tx ∈ rm_i ->
4   exists tx', tx'.(decision) = COMMIT /\ tx.(ops) = tx'.(ops) /\ tx' ∈ tx_i.
```

5 ADVERT

The Advert verification framework includes a Coq implementation of the ADO model and is layered such that ADOs and DApps like those in the previous sections can be reasoned about independently from their protocol-level implementations (Fig. 12). This section discusses how the lower-level implementations relate to their ADO specifications.

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.

Following the approach of previous work [Hawblitzel et al. 2015a; Wilcox et al. 2015], the network is modeled as a logical history (*NetLog*) of events (Fig. 13). The *Send*, *BSend*, *Recv*, and *RecvTO* events correspond, respectively, to sending, broadcasting, receiving, or failing to receive a packet.

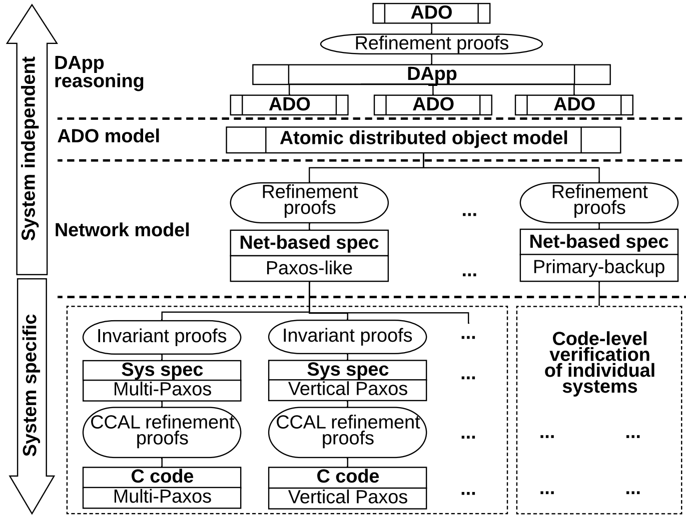


Fig. 12. ADVERT structure.

```
1 param data : Type
2 def time := ℤ * ℕnid
3 def lstate := {cur_tm: time; snap: list data;
4               snap_tm: time}
5 def gstate := ℕnid ⇀ {st: lstate;
6                       tosend: option NetEvent}
7 def phase := Prepare | Write(f: update_func)
8 def msg_kind := Req | Ack
9 def Msg := {kind: (phase * msg_kind); content: lstate}
10 def GMsg := Begin(ph: phase) | End(ph: phase, res: 𝔹)
11 def NetEvent := Send(ℕnid * ℕnid * Msg)
12   | BSend(ℕnid * Msg) | Recv(ℕnid * ℕnid * Msg)
13   | RecvTO(ℕnid) | Ghost(ℕnid * GMsg)
14 def NetLog := list NetEvent
```

```
1 func interpEv (gs: gstate) (ev: NetEvent)
2     : option gstate :=
3   match ev with
4   | Recv(src, dst, {(Prepare, Req), msg}) ⇒
5     (* Haskell-like do-notation *)
6     {st, None} ← gs(dst);
7     if st.cur_tm < msg.cur_tm then
8       let st' := {msg.cur_tm, st.snap,
9                   st.snap_tm} in
10      let ack := {(Prepare, Ack), st'} in
11      gs[dst := {st', Send(dst, src, ack)}]
12    else gs
13  | ... end
14 func interpNet log := fold interpEv log init_gs
```

Fig. 13. Part of the Paxos network specification.

Ghost events do not represent real network communications, but are logical markers to help relate sequences of network events to atomic ADO events (Section 5.2). A network-based specification is a logical state machine whose transitions are triggered by network events. For example, `interpEv` in Fig. 13 shows the transition for a Paxos acceptor upon receiving a `prepare` request.[3]

5.2 Relating Network and ADO Models

We show that the ADO model captures the behavior of the network-based specification by proving a refinement between `pull` and `prepare`, and likewise for `push` and `write`. These theorems state that matching logs of ADO and network events continue to match after taking a step.

```
1 Theorem prepare_pull : forall (ado ado': ADOLog) (net: NetLog),
2   ℝADO ado net -> pull ado = Some ado' ->
3   exists net', prepare net = Some net' /\ ℝADO ado' net'.
```

[3]The implementation in ADVERT is in Coq, but for ease of presentation we use a more concise pseudocode.

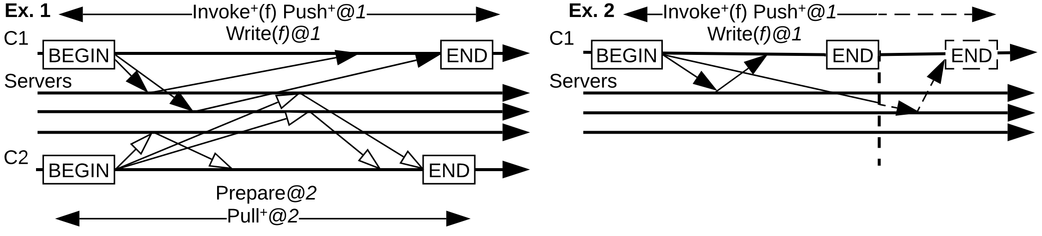


Fig. 14. Asynchronous networks complicate ordering and determining event completion. Examples show Paxos-like systems where successful operations require support from a majority of servers (2 out of 3). Diagonal arrows represent communication in physical time. Dotted arrows represent future events.

The refinement relation $\mathbb{R}_{\text{ADO}}$ establishes a mapping between ADO and network event logs that holds when the replicated state in both models is observably equivalent. This roughly means the methods in the ADO's persistent log and current cache branch match those in the state snapshot carried by the acknowledgement of the client's request with the latest logical timestamp (see Appendix C for precise definitions). The challenge is that the asynchronous network sometimes creates network logs that do not line up cleanly with their corresponding ADO logs. To resolve these mismatches, one must apply certain transformations to the network log and prove that the observable state is preserved. We explain two of these transformations using Paxos-like systems as an example, but similar concepts apply to Chain Replication and Raft as well.

Reordering the Network. Consider **Ex. 1** in Fig. 14 in which C1 tries to commit a function f at logical time 1, while C2 concurrently tries to become an owner at logical time 2. BEGIN and END represent the ghost events emitted before and after the `prepare` and `write` phases. In terms of the physical timeline, C1's `write` phase ends after C2's `prepare` phase; however, the `write` is actually committed as soon as it reaches two servers (a quorum), which happens much earlier. Therefore, in the corresponding ADO logical timeline, $Push^{+}(C1)$ comes before $Pull^{+}(C2)$. To resolve this discrepancy we can reorder the network events such that the entire `write` (everything between BEGIN and END) happens before `prepare` begins. Since f is still committed in the same order the final state does not change. The logs then line up as follows.

$$\begin{array}{lll} Ghost(C1, Begin(Write(f))) & \bullet \cdots \bullet Ghost(C1, End(Write, true)) & \bullet Ghost(C2, Begin(Prepare)) \bullet \cdots \\ Invoke^{+}(C1, f) & \bullet Push^{+}(C1) & \bullet Pull^{+}(C2) \bullet \cdots \end{array}$$

Completing the Network. **Ex. 2** in Fig. 14 illustrates another problem. C1 is again committing f at logical time 1, but this time the request times out after receiving only one acknowledgement (the first END before the vertical dotted line). At this point a quorum has neither accepted nor rejected the request so it is impossible to conclusively say whether it corresponds to $Push^{+}$ or $Push^{-}$. To help make this decision we introduce an abstract *phase scheduler* oracle that determines the future of the network log by modelling clients who arbitrarily call `prepare` and `write`. We can then *complete* C1's `write` by extending the network with the phase scheduler until it has definitively succeeded or failed. In this case, the oracle determines that the delayed request eventually reaches a second server and the `write` succeeds (future events are marked by dotted lines).

Matching Logs. By combining these operations one can transform a log of network events into an equivalent one sorted by logical time where operations have clearly marked beginnings and endings. All that remains is to map sequences of network events to corresponding atomic ADO events. For example, a sequence consisting of $Ghost(C, Begin(Prepare))$, a quorum of positive acknowledgement *Recv* events, and then $Ghost(C, End(Prepare))$ is mapped to $Push^{+}(C)$.

Defining these mappings is simple, but to ensure that the relation is meaningful we must show that matching event logs produce matching states; i.e., a committed method in the ADO's persistent log should also be in a quorum of replicas' local logs. Recall that the ADO model guarantees Replicated State Safety (Section 3.1), which means that committed methods are immutable and are observed by clients in the same order. To demonstrate that the relation between the ADO and network models is valid, we show that the network model satisfies a version of this property (*Network Replicated State Safety*) as well. Note that this is very similar to the top-level safety properties considered by previous work [Hawblitzel et al. 2015a; Ma et al. 2019; Woos et al. 2016].

```
1 Theorem netRepStateSafety : forall (net: NetLog) (gs: gstate),
2   interpNet net = Some gs -> forall (c c': nat),
3     let st := gs(c).(snap) in let st' := gs(c').(snap) in
4     gs(c).(snap_tm) <= gs(c').(snap_tm) -> (* c has a smaller timestamp than c'. Thus its log is a    *)
5     st = firstn (length st) st'.           (* prefix of c'; i.e., c' has all of c's committed methods *)
```

5.3 Safety Proof Template

Proving Network Replicated State Safety is often the most challenging step of the refinement because it requires reasoning at the network level and working with concurrent, non-atomic events. Fortunately, it only needs to be done once per protocol. Furthermore, many distributed protocols are simply 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 subsequent `prepare` and `write` 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. For example, rather than fixing a quorum to be a simple majority, the specification uses an opaque `is_q` function. With some basic assumptions (e.g., quorums have a non-empty intersection), we can build a reusable *proof template* for Network Replicated State 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 proved for free.

Building these proofs of generic, global properties from simple, local invariants is certainly an interesting proof engineering challenge, but one that mostly falls outside the scope and space limitations of this paper. See Appendix D for more details about the proof structure. The following are a few sample instantiations of the parameters, which include the type of the replicated state (`data`), the function to determine if a set of nodes constitutes a quorum (`is_q`), and the update function (`update`), which computes a new value for `data` on a successful `write`.

Paxos [Lamport 2001] uses consensus to replicate a single, immutable value (e.g., an integer). The update function enforces immutability by only accepting the new value if the old state is `None`. Quorums are decided by a simple majority ($f + 1$ out of $2f + 1$) of acknowledgements (`countAcks`). Unlike the other variants, this implements a specialized, write-once version of an ADO.

```
1 def data := option ℤ
2 func update (new: ℤ) (t: time) (old: data) := if old = None then Some(new) else old
3 func is_q (ph: phase) (t: time) (nid: ℕ_nid) (log: NetLog) := countAcks(ph, nid, t, log) ≥ f + 1
```

Multi-Paxos [Renesse and Altinbuken 2015] extends Paxos to a log of immutable values. The log also stores the time at which the value was written. Quorums are the same as in Paxos.

```
1 def data := list (time ∗ ℤ)
2 func update (new: ℤ) (t: time) (old: data) := old ++ [(t, new)]
```

Vertical Paxos [Lamport et al. 2009] permits different quorum sizes between `prepare` and `write` phases. We introduce `conf` to query the configuration (the set of participating servers and the quorum sizes) at a particular logical time. The other parameters are the same as in multi-Paxos.

```
1 param conf : time → (list ℕnid * ℤ * ℤ)
2 func is_q (ph: phase) (t: time) (nid: ℕnid) (log: NetLog) :=
3   let (servs, pquorum, wquorum) := conf t in
4   let quorum := if ph = Prepare then pquorum else wquorum in
5   countAcks(servs, ph, nid, t, log) ≥ quorum
```

CASPaxos [Rystsov 2018] updates the replicated state in-place rather than keeping a log, which eliminates the need for operations like log compaction. Instead of proposing values, clients send "change functions" to compute new states from old ones. For example, a simple compare-and-swap-based lock can be implemented with λ `owner. if owner = None then Some(id) else None` where `id` is the callers' unique ID. Quorums are the same as in Paxos.

```
1 func update (change: data → data) (t: time) (old: data) := change old
```

5.4 Primary Backup and C Code Verification

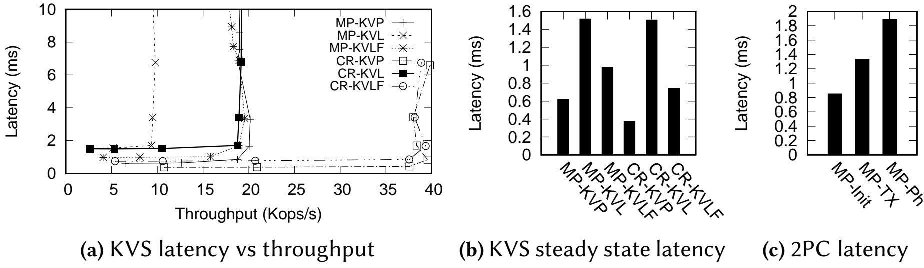
We have focused mainly on Paxos-like systems, but Advert also supports primary backup protocols such as Chain Replication [Renesse and Schneider 2004], and CRAQ [Terrace and Freedman 2009]. Like Paxos these protocols operate in two phases, but rather than using consensus, they ensure consistency by passing updates along a chain of replicas. We have implemented a network-based specification for Chain Replication, and proved that it refines the ADO model. Interestingly, despite the different communication patterns, the proof shares many key elements with Paxos, such as logical time reordering and network completion (Section 5.2).

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]. Compared to the refinement proof for the network-based specification, C code functional correctness proofs are fairly straightforward and follow the approach of CertiKOS [Gu et al. 2016] and WormSpace [Shin et al. 2019]. Together these proofs make it possible to connect ADO specifications of applications like the key-value stores to efficient executable code.

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 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 2020] and runs on both Linux and CertiKOS [Gu et al. 2016] (augmented with unverified `send` and `recv` system calls). The C implementation of Chain Replication is not yet verified, but we expect that doing so would be quite similar to the multi-Paxos case.

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 model in which case the ADO specifications alone are sufficient.



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

Fig. 15. 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)

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. Section 7 discusses a potential integration with Verdi that might also simplify this task. Verifying 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 is abstracted to a network-based model in the style described in Section 5.1. Therefore, although we use CCAL, one could instead choose 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. Fig. 15 shows latency and throughput measurements of C implementations of the key-value store (KVS) and 2PC designs from Section 4. KVS benchmarks use multi-Paxos and Chain Replication while 2PC only uses multi-Paxos. The experiments were run in Amazon EC2 with three acceptors/a three-node chain per ADO for multi-Paxos and Chain Replication respectively. We only vary the write workload, as reads can be optimized with extra learner/cache servers.

For key-value store designs (Figs. 15a and 15b), `KVPrimitive` exhibits the lowest latency and the highest throughput, but cannot separate metadata and data for modularity and manageability. `KVLock`'s lock creates a performance bottleneck as each request accesses it twice (acquire 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 $2f + 1$ nodes, 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. Fig. 15c shows transaction processing latencies of 2PC designs (Fig. 10) 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 more easily experiment with them.

7 RELATED WORK

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 because many distributed protocols naturally split into two phases that map onto `pull` (get the current state and permission to change it) and `push` (commit the changes). The ADO's use of logical event logs to model state machine protocols is also inspired by Mazurkiewic's trace [Mazurkiewicz 1995] and Lamport's c-struct [Lamport 2005]. The network completion operation in the ADO-network refinement is related to a similar notion of completion from work on linearizability of shared-memory concurrent systems in the presence of crashes [Izraelevitz et al. 2016].

Transactions. 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 appends (a prefix of) the working cache tree branch to the persistent log 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, but 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`. An interesting benefit of the ADO model is that these similarities between consensus and transactions are exposed so clearly, and in future work we hope to explore this relation more deeply.

Distributed Object Models. Wang et al. [2019] showed that conflict-free replicated data types (CRDTs) 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 where the ADO model targets strongly consistent systems.

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.

Some programming languages, such as Java [Wollrath et al. 1996], support distributed objects using a remote method invocation protocol. Unlike the ADO model, this is a language-specific

construct, and is designed for concrete implementations rather than abstract specifications. Despite these differences, many key ADO design considerations also apply to Java objects, such as idempotence for methods that may be retried after failure.

Another common object-like abstraction for distributed systems is state machine replication (SMR) and 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 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.

Distributed System Verification. Verdi [Wilcox et al. 2015] is a distributed system verification framework in which one writes an application in a network-based style that is very similar to Advert's network-based specifications (Section 5.1), and reasons about the traces of external events it generates. For example, a verified implementation of Raft in Verdi [Woos et al. 2016] proved its traces are linearizable by showing that it satisfies a property called *State Machine Safety*; i.e., every node in the replicated state machine executes commands in the same order. This is essentially the same as the Network Replicated State Safety property (Section 5.3). One can then reuse this implementation to build applications in an SMR style.

A powerful feature of Verdi is its verified system transformers (VSTs), which can automatically and safely transform a system that assumes a reliable network into one that handles various network faults. Ideally, one could use VSTs to simplify a protocol's linearizability proof at the network level, and then lift it to an ADO specification for application-level reasoning. As a step in this direction, we have begun a proof that Verdi's network-based specification of Raft refines the ADO model.

The main challenge in this proof is reconciling the different network specification styles. For example, our Paxos network-based specification emits `Begin` and `End` ghost events, which mark when a node performs an internal transition. On a `Begin` event a node increments its local timestamp in preparation for a new round of updates, and at `End` it copies the state snapshot from the acknowledgement with the largest logical timestamp. These are then used to reorder and complete the network in the refinement with the ADO model (see Section 5.2 and Appendix C). Verdi's Raft implementation has similar internal transitions, but they are bundled with transitions for physical packets such as `ClientRequest` or `Acknowledgement` rather than using separate ghost events. This is not a fundamental incompatibility, and can be solved by introducing an additional intermediate specification with extra ghost events and proving it equivalent to the Verdi specification.

IronFleet [Hawblitzel et al. 2015a] is another framework in which distributed systems are also modeled as network-based state machines. One then uses reduction arguments to reorder, remove, or join sequences of network events into simpler ones (e.g., commuting unrelated sends and receives). This is similar to the network reordering step in the ADO refinement proofs. In principle one could connect IronFleet's techniques with Advert's ADO model in a similar manner to Verdi, but there is a technical incompatibility due to the different proof assistants used by each framework (Dafny [Leino 2010] vs. Coq [The Coq Development Team 2018]).

There is also a large body of work on techniques for transforming an asynchronous program into into an equivalent sequential one by 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 node 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. Other projects [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 distributed system abstraction; however, they could simplify or automate certain network-level proofs in Advert. For example, Padon et al. [2017] automatically verifies the safety of (first-order logic models of) a variety of Paxos variants in Ivy, which is an important step in the ADO refinement proofs.

Distributed System 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), which define how clients can interact with a set of ADOs.

Disel [Sergey et al. 2017] is another framework that supports decomposing composite systems into isolated pieces. Distributed 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 define what communication is allowed and what preconditions 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 patterns of sends and receives rather than abstract method calls.

Another line of work on distributed system composition using a separation logic is Aneris [Krogh-Jespersen et al. 2020]. It provides a higher-order ML-like language for writing specifications and the program logic is built on top of Iris. Notably, it supports node-level concurrency through multi-threading in addition to the concurrency from the distributed nodes. 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 involving a coordinator, such as Two-Phase Commit, but to our knowledge they have not fully demonstrated support for more decentralized applications such as `KVLockFree`.

Exploiting Exposed Failures and Intermediate States. 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 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 can 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. 2005] 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 makes formal verification of such systems much more feasible, which could encourage their development and adoption.

8 CONCLUSIONS

The atomic distributed object model is a compositional abstraction for reasoning about strongly consistent distributed systems 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. We believe the ADO model is a powerful tool for developing efficient, bug-free distributed applications and opens many exciting avenues for future study.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful feedback. This material is based upon work supported in part by NSF grants 2019285, 1763399, and 1521523, and by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-21-C-4018. The fourth author is a co-founder of and has an equity interest in CertiK Global Ltd. CertiK has licensed Yale University's intellectual property, which is related to the NSF grants 1521523 and 1763399.

REFERENCES

Andrew W. Appel. 2011. Verified Software Toolchain. In *Proc. of the 20th European Symposium on Programming* (Saabruücken, Germany) *(ESOP '11, Vol. 6602)*, Gilles Barthe (Ed.). Springer, Berlin, Heidelberg, 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. 2012. CORFU: A Shared Log Design for Flash Clusters. In *Proc. of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA, USA) *(NSDI '12)*. USENIX Association, Berkeley, CA, USA, 1–14. https://doi.org/10.1145/2535930

Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. 2003. Deconstructing Paxos. *SIGACT News* 34, 1 (March 2003), 47–67. https://doi.org/10.1145/637437.637447

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

Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming* (2nd ed.). Springer-Verlag, Berlin, Heidelberg.

Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2019. The Rise of Serverless Computing. *Commun. ACM* 62, 12 (2019), 44–54. https://doi.org/10.1145/3368454

Tej Chajed, Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. 2018. Verifying Concurrent Software Using Movers in CSPEC. In *Proc. of the 13th USENIX Symposium on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI '18)*. USENIX Association, Carlsbad, CA, 306–322. 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. 2006. Bigtable: A Distributed Storage System for Structured Data. In *Proc. of the 7th USENIX Symposium on Operating Systems Design and Implementation* (Seattle, WA, USA) *(OSDI '06)*. ACM, New York, NY, USA, 205–218. https://doi.org/10.1145/1365815.1365816

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

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

Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. 2018. Compositional Programming and Testing of Dynamic Distributed Systems. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 1–30. https://doi.org/10.1145/3276529

Pascal Felber, Ben Jai, Rajeev Rastogi, and Mark Smith. 2001. Using Semantic Knowledge of Distributed Objects to Increase Reliability and Availability. In *Proc. of the 6th International Workshop on Object-Oriented Real-Time Dependable Systems* (Rome, Italy) *(WORDS '01)*. IEEE Computer Society, Washington, DC, USA, 153–160. https://doi.org/10.1109/WORDS.2001.945126

Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. 2010. Abstraction for Concurrent Objects. *Theor. Comput. Sci.* 411, 51-52 (2010), 4379–4398. https://doi.org/10.1007/978-3-642-00590-9_19

Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proc. of the 12th European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. ACM, New York, NY, USA, 328–343. https://doi.org/10.1145/3064176.3064183

Eli Gafni and Leslie Lamport. 2003. Disk Paxos. *Distributed Computing* 16, 1 (2003), 1–20. https://doi.org/10.1007/s00446-002-0070-8

Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *Proc. of the ACM SIGCOMM 2011 Conference*. ACM, New York, NY, USA, 350–361. https://doi.org/10.1145/2043164.2018477

Jim Gray and Leslie Lamport. 2006. Consensus on Transaction Commit. *ACM Transactions on Database Systems* 31, 1 (2006), 133–160. 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. 2015. Deep Specifications and Certified Abstraction Layers. In *Proc. of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. ACM, New York, NY, USA, 595–608. https://doi.org/10.1145/2676726.2676975

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI '16)*. USENIX Association, Berkeley, CA, USA, 653–669. 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. 2018. Certified Concurrent Abstraction Layers. In *Proc. of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI '18)*. ACM, New York, NY, USA, 646–661. https://doi.org/10.1145/3296979.3192381

Rachid Guerraoui and Michal Kapalka. 2008. On the Correctness of Transactional Memory. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Salt Lake City, UT, USA) *(PPoPP '08)*. ACM, New York, NY, USA, 175–184. 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. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proc. of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '14)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/2670979.2670986

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

Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-end Security via Automated Full-system Verification. In *Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation* (Broomfield, CO, USA) *(OSDI '14)*. USENIX Association, Berkeley, CA, USA, 165–181. https://dl.acm.org/doi/10.5555/2685048.2685062

Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serar Tasiran. 2015b. Automated and Modular Refinement Reasoning for Concurrent Programs. In *Proc. of the 27th International Conference on Computer Aided Verification* (San Francisco, CA, USA) *(CAV '15)*. Springer-Verlag, Berlin, Heidelberg, 449–465. https://doi.org/10.1007/978-3-319-21668-3_26

Wolf Honoré, Jieung Kim, Ji-Yong Shin, and Zhong Shao. 2021. *Much ADO about Failures: A Fault-Aware Model for Compositional Verification of Strongly Consistent Distributed Systems.* Technical Report YALEU/DCS/TR-1557. Yale Univ. https://flint.cs.yale.edu/publications/ado.html

Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Proc. of the 30th International Symposium on Distributed Computing* (Paris, France) *(DISC '16)*. Springer-Verlag, Berlin, Heidelberg, 313–327. https://doi.org/10.1007/978-3-662-53426-7_23

Tom Killalea. 2016. The Hidden Dividends of Microservices. *Commun. ACM* 59, 8 (July 2016), 42–45. https://doi.org/10.1145/2948985

Eric Koskinen and Matthew Parkinson. 2015. The Push/Pull Model of Transactions. In *Proc. of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 186–195. https://doi.org/10.1145/2737924.2737995

Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. 2020. Inductive Sequentialization of Asynchronous Programs. In *Proc. of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI '20)*. ACM, New York, NY, USA, 227–242. https://doi.org/10.1145/3385412.3385980

Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *Proc. of the 27th European Symposium on Programming (ESOP '20)*, Peter Müller (Ed.). Springer-Verlag, Berlin, Heidelberg, 336–365. https://doi.org/10.1007/978-3-030-44914-8_13

Leslie Lamport. 2001. Paxos Made Simple. *SIGACT News* 32, 4 (Dec. 2001), 51–58. https://doi.org/10.1145/568425.568433

Leslie Lamport. 2005. *Generalized Consensus and Paxos.* Technical Report MSR-TR-2005-33. Microsoft.

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

Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2009. Vertical Paxos and Primary-Backup Replication. In *Proc. of the 28th ACM Symposium on Principles of Distributed Computing* (Calgary, AB, Canada) *(PODC '09)*. ACM, New York, NY, USA, 312–313. https://doi.org/10.1145/1582716.1582783

Butler Lampson. 2001. The ABCD's of Paxos. In *Proc. of the 20th Annual ACM Symposium on Principles of Distributed Computing* (Newport, RI, USA) *(PODC '01)*. ACM, New York, NY, USA, 13. https://doi.org/10.1145/383962.383969

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

Xavier Leroy. 2005–2020. The CompCert verified compiler. http://compcert.inria.fr/.

Xavier Leroy. 2009. A Formally Verified Compiler Back-End. *Journal of Automated Reasoning* 43, 4 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

Hongjin Liang, Jan Hoffmann, Xinyu Feng, and Zhong Shao. 2013. Characterizing Progress Properties of Concurrent Objects via Contextual Refinements. In *Proc. of the 24th International Conference on Concurrency Theory* (Buenos Aires, Argentina) *(CONCUR '13)*. Springer-Verlag, Berlin, Heidelberg, 227–241. https://doi.org/10.1007/978-3-642-40184-8_17

Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. 2019. I4: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols. In *Proc. of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, ON, Canada) *(SOSP '19)*. ACM, New York, NY, USA, 370–384. https://doi.org/10.1145/3341301.3359651

John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. 2004. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *Proc. of the 6th USENIX Symposium on Operating Systems Design and Implementation* (San Francisco, CA, USA) *(OSDI '04, Vol. 4)*. USENIX Association, Berkeley, CA, USA, 8–8. https://dl.acm.org/doi/10.5555/1251254.1251262

Antoni Mazurkiewicz. 1995. *Introduction to Trace Theory.* World Scientific, Hackensack, NJ, USA, Chapter 1, 3–41. https://doi.org/10.1142/9789814261456_0001

Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. 2018. A Large Scale Study of Data Center Network Reliability. In *Proc. of the Internet Measurement Conference* (Boston, MA, USA) *(IMC '18)*. ACM, New York, NY, USA, 393–407. https://doi.org/10.1145/3278532.3278566

Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proc. of the 24th ACM Symposium on Operating Systems Principles* (Farminton, PA, USA) *(SOSP '13)*. ACM, New York, NY, USA, 358–372. https://doi.org/10.1145/2517349.2517350

Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. 2005. Speculative Execution in a Distributed File System. *ACM SIGOPS Operating Systems Review* 39, 5 (2005), 191–205. https://doi.org/10.1145/1189256.1189258

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

Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017. Paxos Made EPR: Decidable Reasoning About Distributed Protocols. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 108 (Oct. 2017), 31 pages. https://doi.org/10.1145/3140568

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

Ganesan Ramalingam and Kapil Vaswani. 2013. Fault Tolerance via Idempotence. In *Proc. of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. ACM, New York, NY, USA, 249–262. https://doi.org/10.1145/2429069.2429100

Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos Made Moderately Complex. *ACM Computing Surveys (CSUR)* 47, 3 (2015), 42. https://doi.org/10.1145/2673577

Robbert Van Renesse and Fred B. Schneider. 2004. Chain Replication for Supporting High Throughput and Availability. In *Proc. of the 6th USENIX Symposium on Operating Systems Design and Implementation* (San Francisco, CA, USA) *(OSDI '04, Vol. 4)*. USENIX Association, Berkeley, CA, USA, 91–104. https://dl.acm.org/doi/10.5555/1251254.1251261

Denis Rystsov. 2018. CASPaxos: Replicated State Machines without Logs. *CoRR* abs/1802.07000 (2018), 15 pages. arXiv:1802.07000 http://arxiv.org/abs/1802.07000

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *Proc. of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*. ACM, New York, NY, USA, 158–174. https://doi.org/10.1145/3453483.3454036

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

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2017. Programming and Proving with Distributed Protocols. *Proc. ACM Program. Lang.* 2, POPL, Article 28 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158116

Ji-Yong Shin, Jieung Kim, Wolf Honoré, Hernán Vanzetto, Srihari Radhakrishnan, Mahesh Balakrishnan, and Zhong Shao. 2019. WormSpace: A Modular Foundation for Simple, Verifiable Distributed Systems. In *Proc. of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. ACM, New York, NY, USA, 299–311. https://doi.org/10.1145/3357223.3362739

Vilhelm Sjöberg, Yuyang Sang, Shu chun Weng, and Zhong Shao. 2019. DeepSEA: A Language for Certified System Software. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 136 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360562

Andrew S. Tanenbaum and Maarten van Steen. 2006. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Singapore. 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. 2018. Modularity for Decidability: Implementing and Semi-Automatically Verifying Distributed Systems. In *Proc. of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI '18)*. ACM, New York, NY, USA, 662–677. https://doi.org/10.1145/3192366.3192414

Jeff Terrace and Michael J. Freedman. 2009. Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads. In *USENIX Annual Technical Conference*. San Diego, CA, USENIX Association, Berkeley, CA, USA, 16 pages. https://dl.acm.org/doi/10.5555/1855807.1855818

The AWS Team. 2011. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. https://aws.amazon.com/message/65648/.

The Coq Development Team. 1999–2018. The Coq Proof Assistant. http://coq.inria.fr.

Ben Treynor. 2011. Today's Outage for Several Google Services. 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. 2019. Pretend Synchrony: Synchronous Verification of Asynchronous Distributed Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 59 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290372

Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. 1994. *A Note on Distributed Computing*. Technical Report. IEEE Micro. https://dl.acm.org/doi/10.5555/974938

Chao Wang, Constantin Enea, Suha Orhun Mutluergil, and Gustavo Petri. 2019. Replication-Aware Linearizability. In *Proc. of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI '19)*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, New York, NY, USA, 980–993. https://doi.org/10.1145/3314221.3314617

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proc. of the 36th ACM SIGPLAN*

Conference on Programming Language Design and Implementation (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 357–368. https://doi.org/10.1145/2737924.2737958

Ann Wollrath, Roger Riggs, and Jim Waldo. 1996. A Distributed Object Model for the Java System. *Comput. Syst.* 9 (1996), 265–290. https://dl.acm.org/doi/10.5555/1268049.1268066

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

Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *Proc. of the 25th Symposium on Operating Systems Principles* (Monterey, CA, USA) *(SOSP '15)*. ACM, New York, NY, USA, 263–278. https://doi.org/10.1145/2815400.2815404

A ADO FORMAL SEMANTICS

Due to space limitations, Section 2.4 focused on the outcomes of `pull` and `push` that have an interesting effect on the distributed state and omitted certain total failure cases. For completeness this section presents the full ADO semantics including the previously omitted cases.

State Definitions. The state definitions are the same as before.

$$
\begin{aligned}
ADO^{\Sigma} &\triangleq \Sigma * (\exists R.\, Method \rightharpoonup (\Sigma \to \Sigma * R)) \\
CID &\triangleq \langle \mathbb{N}_{nid} * \mathbb{N}_{time} * CID \rangle \mid \mathbf{Root} \\
Cache &\triangleq CID * Method \\
PersistLog &\triangleq List(Cache) \\
CacheTree &\triangleq Set(Cache) \\
CIDMap &\triangleq \mathbb{N}_{nid} \rightharpoonup CID \\
OwnerMap &\triangleq \mathbb{N}_{time} \rightharpoonup (\mathbb{N}_{nid} \mid \mathbf{NoOwn}) \\
DSM &\triangleq PersistLog * CacheTree * CIDMap * OwnerMap
\end{aligned}
$$

$$
\begin{aligned}
ADOEvent \triangleq\ & Pull^{+}(\mathbb{N}_{nid} * \mathbb{N}_{time} * CID) \\
\mid\ & Pull^{*}(\mathbb{N}_{nid} * \mathbb{N}_{time}) \\
\mid\ & Pull^{-}(\mathbb{N}_{nid}) \\
\mid\ & Invoke^{+}(\mathbb{N}_{nid} * Method) \\
\mid\ & Invoke^{-}(\mathbb{N}_{nid}) \\
\mid\ & Push^{+}(\mathbb{N}_{nid} * CID) \\
\mid\ & Push^{-}(\mathbb{N}_{nid}) \\
ADOLog \triangleq\ & List(ADOEvent)
\end{aligned}
$$

Log Generation. The log generation rules include three additional cases: GenPullFailure, GenMethodFailure, and GenPushFailure. The `pull` and `push` cases can occur at any time if the oracle returns *Fail*. A method invocation fails if the caller's active *CID* is not in the cache tree. This prevents clients from calling methods without first calling `pull`, since that is the only way for an active *CID* to be assigned. It also ensures that clients cannot continue working on stale states that have been invalidated by a successful `push` (much how multi-Paxos acceptors and Raft followers reject updates with out-of-date ballot/term numbers).

GenPullSuccess

$$
\frac{\mathbb{O}_{pull}(log, nid) = Ok(time, cid) \qquad timeOf(cid) < time \qquad noOwnerAt(log, time) \qquad (cid \in caches(log) \lor cid = root(log))}{\mathbb{O}_{pull} \vdash pull(nid) : log \longrightarrow log \bullet Pull^{+}(nid, time, cid)}
$$

GenPullPreempt

$$
\frac{\mathbb{O}_{pull}(log, nid) = Preempt(time) \qquad time \notin dom(owners(log))}{\mathbb{O}_{pull} \vdash pull(nid) : log \longrightarrow log \bullet Pull^{*}(nid, time)}
$$

GenPullFailure

$$
\frac{\mathbb{O}_{pull}(log, nid) = Fail}{\mathbb{O}_{pull} \vdash pull(nid) : log \longrightarrow log \bullet Pull^{-}(nid)}
$$

GenMethodInvocation

$$
\frac{cids(log)[nid] \in caches(log)}{\vdash M(nid) : log \longrightarrow log \bullet Invoke^{+}(nid, M)}
$$

GenMethodFailure

$$
\frac{cids(log)[nid] \notin caches(log)}{\vdash M(nid) : log \longrightarrow log \bullet Invoke^{-}(nid)}
$$

GenPushSuccess

$$
\frac{\mathbb{O}_{push}(log, nid) = Ok(ccid) \qquad nidOf(ccid) = maxOwner(log) = nid \qquad timeOf(ccid) = timeOf(cids(log)[nid]) \qquad ccid \in caches(log)}{\mathbb{O}_{push} \vdash push(nid) : log \longrightarrow log \bullet Push^{+}(nid, ccid)}
$$

GenPushFailure

$$
\frac{\mathbb{O}_{push}(log, nid) = Fail}{\mathbb{O}_{push} \vdash push(nid) : log \longrightarrow log \bullet Push^{-}(nid)}
$$

$$\begin{aligned}
\mathbb{O}_{pull} &: ADOLog \to \mathbb{N}_{nid} \to (Ok(\mathbb{N}_{time} * CID) \mid Preempt(\mathbb{N}_{time}) \mid Fail) \\
\mathbb{O}_{push} &: ADOLog \to \mathbb{N}_{nid} \to (Ok(CID) \mid Fail) \\
nidOf(cid) &\triangleq \textbf{let } \langle nid, _, _ \rangle = cid \textbf{ in } nid \\
timeOf(cid) &\triangleq \textbf{let } \langle _, time, _ \rangle = cid \textbf{ in } time \\
root(log) &\triangleq \textbf{let } (p, _, _, _) = interpADO(log) \textbf{ in if } p \neq [] \textbf{ then } last(p) \textbf{ else Root} \\
caches(log) &\triangleq \textbf{let } (_, cs, _, _) = interpADO(log) \textbf{ in } cs \\
cids(log) &\triangleq \textbf{let } (_, _, cids, _) = interpADO(log) \textbf{ in } cids \\
owners(log) &\triangleq \textbf{let } (_, _, _, owns) = interpADO(log) \textbf{ in } owns \\
noOwnerAt(log, time) &\triangleq time \notin dom(owners(log)) \vee owners(log)[time] = \textbf{NoOwn} \\
maxOwner(log) &\triangleq \textbf{let } owns = owners(log) \textbf{ in } owns[max(dom(owns))]
\end{aligned}$$

Log Interpretation. The log interpretation rules also include three new cases: PullFailure, MethodFailure, and PushFailure. Each of these cases represents a total failure that has no effect on the distributed state. One might then question why these failures are recorded in the log of events at all. The reason is that they help to model the passing of time in the real world. In reality a `pull` might initially fail because the network is temporarily down, but then succeed if it is called again after a few seconds, even if nothing else in the system changes. Recall that the oracles $\mathbb{O}_{pull}$ and $\mathbb{O}_{push}$ are deterministic functions, so if $\mathbb{O}_{pull}(log, nid) = Fail$ then it will always return *Fail* if *log* and *nid* do not change. By appending $Pull^-$ to the log, GenPullFailure makes it possible for a client to observe different results when calling `pull` twice in a row.

PullSuccess
$$\frac{cids' = cids[nid \mapsto \langle nid, time, cid \rangle] \qquad owns' = voteNoOwn(owns[time \mapsto nid], time - 1)}{Pull^+(nid, time, cid) : (p, cs, cids, owns) \longrightarrow (p, cs, cids', owns')}$$

PullPreempt
$$\frac{owns' = voteNoOwn(owns, time)}{Pull^*(nid, time) : (p, cs, cids, owns) \longrightarrow (p, cs, cids, owns')}$$

PullFailure
$$\frac{}{Pull^-(nid) : (p, cs, cids, owns) \longrightarrow (p, cs, cids, owns)}$$

MethodInvocation
$$\frac{cs' = cs \cup \{(cids[nid], M)\} \qquad cids' = cids[nid \mapsto nextCID(cid)]}{Invoke^+(nid, M) : (p, cs, cids, owns) \longrightarrow (p, cs', cids', owns)}$$

MethodFailure
$$\frac{}{Invoke^-(nid) : (p, cs, cids, owns) \longrightarrow (p, cs, cids, owns)}$$

PushSuccess
$$\frac{(\vec{c}_{ok}, cs') = partition(cs, ccid)}{Push^+(nid, ccid) : (p, cs, cids, owns) \longrightarrow (p \bullet \vec{c}_{ok}, cs', cid, owns)}$$

PushFailure
$$\frac{}{Push^-(nid) : (p, cs, cids, owns) \longrightarrow (p, cs, cids, owns)}$$

$$
\begin{aligned}
cid_1 < cid_2 &\triangleq cid_2 \neq \textbf{Root} \land \textbf{let } \langle _, _, parent \rangle = cid_2 \textbf{ in } cid_1 = parent \lor cid_1 < parent \\
cid_1 \leq cid_2 &\triangleq cid_1 < cid_2 \lor cid_1 = cid_2 \\
voteNoOwn(owns, time) &\triangleq owns[t \mapsto \textbf{NoOwn} \mid \forall t \leq time.\, t \notin dom(owns)] \\
nextCID(cid) &\triangleq \textbf{let } \langle nid, time, _ \rangle = cid \textbf{ in } \langle nid, time, cid \rangle \\
partition(cs, cid) &\triangleq \textbf{let } \vec{c}_{ok} = sort(\{(c, M) \in cs \mid c \leq cid\}) \textbf{ in} \\
&\quad\ \textbf{let } cs' = \{(c, M) \in cs \mid cid < c\} \textbf{ in } (\vec{c}_{ok}, cs')
\end{aligned}
$$

B MORE ADO EXAMPLES

Some of the ADO examples in Section 4 intentionally make some simplifying assumptions (e.g., ignoring liveness or certain types of failures) for the sake of a cleaner presentation. In this section we show that one can remove these assumptions to create objects that are closer to those used in real-world systems.

B.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. Fig. 16 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 node's current ticket. A node owns the lock when its ticket matches the one being served. The owner can then release the lock by incrementing the serving ticket, which implicitly causes the next waiting node to acquire the lock. The waiting nodes implicitly form a queue, which provides better fairness than `CASLock` because nodes 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 node'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 node 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 node to acquire the lock and break mutual exclusivity. However, due to distributed systems' lack of a

```
1 ADO TicketLock {
2   shared next : ℕ := 0;
3   shared serving : ℕ := 0;
4   shared tkts : Map[ℕ ⇀ ℕ] := {};
5   method requestTkt() {
6     if (this.nid ∈ this.tkts) { return this.tkts[this.nid]; }
7     else {
8       this.next += 1;
9       this.tkts[this.nid] := this.next;
10      return this.next; } }
11  method tryAcquire() {
12    return this.nid ∈ this.tks && this.serving = this.tkts[this.nid]; }
13  method release() {
14    if (this.nid ∈ this.tks && this.serving = this.tkts[this.nid]) {
15      this.serving += 1;
16      this.tkts.delete(this.nid); } } }
```

```
1 ADO TimeoutLock {
2   shared owner : option ℕ := None;
3   shared time : ℕ := 0;
4   method tryAcquire() {
5     if (this.owner = None) {
6       this.owner := Some(this.nid);
7       this.time := 0; }
8     return this.owner = Some(this.nid); }
9   method release() {
10    if (this.owner = Some(this.nid)) { this.owner := None; } }
11  method checkin() {
12    if (this.owner = Some(this.nid)) { this.time := 0; }
13    return this.owner = Some(this.nid); }
14  method tick() {
15    this.time += 1;
16    if (this.time > TIMEOUT) { this.owner := None; } } }
```

Fig. 16. More complex ADO locks.

synchronized global clock, a common assumption is that the nodes' local clocks are within some error bound so this situation cannot occur [Cachin et al. 2011; Hawblitzel et al. 2015a].

B.2 2PC with Recovery

The 2PC example in Section 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.

Fig. 17 shows the same `TM` and `RM` from Fig. 10, 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

```
1 ADO RM {
2   shared txs : Vector[TX] := [];
3   method prepare(tx) { ... }
4   method decide(ts, decision) { ... }
5   method remove_tx_after(ts) { this.txs := this.txs.filter(λ tx. tx.ts ≤ ts); }
6   method read() { return this.txs; } }
```

```
1 DApp TM(rm_1: RM, ..., rm_n: RM) {
2   local ts : ℤ := 0;
3   /* Must be called once when TM starts */
4   proc init() {
5     for rm in [this.rm_1, ..., this.rm_n] { while (rm.pull() = FAIL) {} }
6     /* Recovery */
7     need_recovery := [];
8     /* Check the last decided TX in each RM */
9     for rm in [this.rm_1, ..., this.rm_n] {
10      rm.read();
11      do { txs := rm.push(); } while (txs = FAIL);
12      need_recovery[rm] = txs.last().decision ∉ [COMMIT, ABORT];
13      last_decided := txs.filter(λ tx. tx.decision ∈ [COMMIT, ABORT]).last();
14      /* Remember the timestamp and decision of the latest decided TX */
15      if (last_decided.ts > this.ts) {
16        this.ts := last_decided.ts;
17        decision := last_decided.decision; } }
18    /* All RMs are in consistent state */
19    if (!need_recovery.any()) { return; }
20    for rm in [this.rm_1, ..., this.rm_n] {
21      /* Finish TXs that failed during phase 2 */
22      if (need_recovery[rm]) { rm.decide(this.ts, decision); }
23      /* Wipe out TXs that failed during phase 1 */
24      rm.remove_tx_after(this.ts);
25      while (rm.push() = FAIL) { } } }
26  proc handle_request(ops) { ... } }
```

Fig. 17. Two-Phase Commit with recovery.

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 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.

C ADO REFINEMENT FORMAL DEFINITIONS

In this section we present the definition of $\mathbb{R}_{\text{ADO}}$ and several important auxiliary definitions.

Definition 1 (Phase Scheduler). *A* phase schedule *is a list of Prepare and Write events indicating the order that a certain client will call the corresponding operations. A* phase scheduler, $\mathbb{O}_{sched}$, *is an oracle that takes a NetLog and a client ID and returns a phase schedule.*

Definition 2 (Logical Time at an Event). *Given a NetLog λ, a NetEvent ε, and a server S, where $\lambda = \lambda' \bullet \varepsilon \bullet _$, S's logical time at ε is t ($time_S(\lambda, \varepsilon) = t$) if $\exists gs.\, Some(gs) = interpNet(\lambda' \bullet \varepsilon) \wedge t = gs(S).st.cur_tm$.*

Definition 3 (Acknowledges). *Given a NetLog λ, a server S* acknowledges *a request by a client C if λ contains a $Recv(C, S, _)$ event followed by $Send(S, C, _)$. The acknowledgements of an operation by a client C at logical time t is the set of $Recv(C, _, msg)$ events that occur after some $\varepsilon = Ghost(C, Begin(_))$ event where $t = msg.content.cur_tm = time_C(\lambda, \varepsilon)$.*

Definition 4 (Complete Operation). *A NetLog λ has a* complete operation *by a client C at logical time t if there is a $\varepsilon = Ghost(C, Begin(_))$ event in λ such that $t = time_C(\lambda, \varepsilon)$ and a quorum of servers has either acknowledged or rejected the request. The operation is* incomplete *if no quorum has yet acknowledged or rejected the request.*

Definition 5 (Completion). *Given a NetLog λ, a client C, a logical time t, and a phase scheduler $\mathbb{O}_{sched}$, a completion up to t ($complete_{t,C}(\lambda)$) computes a new log of future NetEvents by repeatedly applying either `prepare` or `write` according to the schedule generated by $\mathbb{O}_{sched}$ until C's operation at t is complete. A completion ($complete(\lambda)$) computes $complete_{t,C}(\lambda)$ for all incomplete operations by C at t in λ.*

Definition 6 (Logical Time Reordering). *A NetLog λ's* logical time reordering *($reorder(\lambda)$) is computed by sorting the events of λ by logical time with the following caveats: events originating from the same node remain in program order, Recv events cannot be moved before their corresponding Send or BSend events, and once a server receives an Recv event with logical time t, any later Recv events at the same server with logical time less than t are dropped instead of being reordered.*

Definition 7 (Well-Bracketed Network). *A NetLog λ is* well-bracketed *with respect to client C when it is at the end of a phase and C is allowed to continue issuing updates; i.e., $\lambda = _ \bullet Ghost(C, End(Prepare, true))$ or $\lambda = _ \bullet Ghost(C, End(Write, _))$.*

Definition 8 (Local View Match). *The* local view *of NetLog λ by a client C at logical time t is the snapshot carried by an acknowledgement ack for C's latest complete operation such that $ack.snapshot.snap_tm$ is the largest among the acknowledgments and is less than or equal to t. The local view of a DSM is the persistent state plus the current cache branch; i.e., for $(p, cs, cids, owns)$ it is $p \bullet fst(partition(cs, cid[C]))$. A NetLog λ and a DSM Δ have matching local views ($\mathbb{R}_{\text{view}}$) if the value computed by replaying the events from Δ's local view equals the last value in λ's local view.*

Definition 9 ($\mathbb{R}_{\text{ADO}}$). *Given a NetLog λ_{net} and an ADOLog λ_{ADO} where C is an owner, the refinement relation $\mathbb{R}_{\text{ADO}}(\lambda_{net}, \lambda_{ADO})$ holds when λ_{net} is well-bracketed with respect to C, and $\mathbb{R}_{\text{view}}(reorder(complete(\lambda_{net}))$*

D NETWORK SAFETY PROOF TEMPLATE

This section explains the network based specification for Paxos-like protocols in more detail and shows a few representative theorems from the Paxos proof template. Figs. 18 and 19 show more of the network based specification and Fig. 20 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.

State Machine. Every node maintains local state (`lstate`) that consists of its view of the current logical time (an integer plus a node ID to act as a tie breaker), a snapshot of the history of the replicated data, and the time that the snapshot was taken. For a concrete implementation of this

```
1 param data : Type
2 def time := ℤ ∗ ℕnid
3 def lstate := {cur_tm: time; snap: list data; snap_tm: time}
4 def gstate := ℕnid ⇀ {st: lstate; tosend: option NetEvent}
5 def update_func := time → data → data
6 def phase := Prepare | Write(f: update_func)
7 def msg_kind := Req | Ack
8 def Msg := {kind: (phase ∗ msg_kind); content: lstate}
9 def GMsg := Begin(ph: phase) | End(ph: phase, res: 𝔹)
10 def NetEvent := Send(ℕnid ∗ ℕnid ∗ Msg)
11               | BSend(ℕnid ∗ Msg)
12               | Recv(ℕnid ∗ ℕnid ∗ Msg)
13               | RecvTO(ℕnid)
14               | Ghost(ℕnid ∗ GMsg)
15 def NetLog := list NetEvent
16
17 func interpEv (gs: gstate) (ev: NetEvent) : option gstate :=
18   match ev with
19   | Recv(src, dst, {(Prepare, Req), msg}) ⇒
20     {st, None} ⟵ gs(dst); (* Haskell-like do-notation *)
21     if st.cur_tm < msg.cur_tm then
22       let st' := {msg.cur_tm, st.snap, st.snap_tm} in
23       gs[dst := {st', Send(dst, src, {(Prepare, Ack), st'})}]
24     else gs
25   | Recv(src, dst, {(Write(f), Req), msg}) ⇒
26     {st, None} ⟵ gs(dst);
27     if st.cur_tm = msg.cur_tm then
28       let snap' := msg.snap • f(msg.cur_tm, last(msg.snap)) in
29       let st' := {msg.cur_tm, snap', msg.cur_tm} in
30       gs[dst := {st', Send(dst, src, {(Write(f), Ack), st'})}]
31     else gs
32   | Ghost(src, Begin(Prepare)) ⇒
33     {st, None} ⟵ gs(src);
34     let st' := {(fst(st.cur_tm) + 1, snd(st.cur_tm)),
35                 init_data, (0, snd(st.cur_tm))} in
36     gs[src := {st', BSend(src, {(Prepare, Req), st'})}]
37   | ... end
38
39 func interpNet log := fold interpEv log init_gstate
```

Fig. 18. Paxos-like network log interpretation.

specification it is sufficient to maintain only the latest data rather than its history, but storing the history as ghost state helps in defining the relation to the ADO model. The global state (`gstate`) of the system is a map from each node's ID to a pair of its local state and the next message it intends to send. A message (`Msg`) carries a local state, an indication of which phase it was sent in, and whether it is a client request or a server acknowledgement (`msg_kind`). Ghost messages mark the beginnings and ends of phases. The `End` event also carries a flag stating whether the operation succeeded in gathering acknowledgements from a quorum of servers. The `f` argument in the `Write`

```
1 param nid : ℕnid
2 param servers : list ℕnid
3 param is_q : phase ⟶ time ⟶ NetLog ⟶ 𝔹
4
5 (* Adds sequence of events ending in either Recv or RecvTO *)
6 func recv (log: NetLog) : NetLog := ...
7 (* Wait for a reply from each server *)
8 func recvAll (log: NetLog) : NetLog := fold recv servers log
9
10 (* Client functions *)
11 func sendReq (ph: phase) (log: NetLog) : option NetLog :=
12   let log' := log • Ghost(nid, Begin(ph)) in
13   gs ⟵ interpNet(log');
14   {_, BSend(nid, msg)} ⟵ gs(nid);
15   let log'' := recvAll(log' • BSend(nid, msg)) in
16   gs' ⟵ interpNet(log'');
17   let res := is_q(ph, gs'(nid).st.cur_tm, log'') in
18   log'' • Ghost(nid, End(ph, res))
19
20 func prepare (log: NetLog) := sendReq(Prepare, log)
21 func write (f: update_func) (log: NetLog) :=
22   _ • Ghost(nid, End(Prepare, true)) ⟵ log;
23   sendReq(Write(f), log)
24
25 (* Server function *)
26 func ackReq (log: NetLog) : option NetLog :=
27   let log' := recv(log) in
28   gs ⟵ interpNet(log');
29   {_, tosnd} ⟵ gs(nid);
30   if tosnd = Send(nid, dst, msg) then log' • Send(nid, dst, msg)
31   else log'
```

Fig. 19. Paxos-like network-based specifications.

phase is known as the *update function*. This is a per-protocol function that is used to compute the value to replicate in each round and is one of the parameters that makes this specification generic.

Fig. 18 shows three representative cases for interpreting different network events. The first two show how a server handles a request from a client in both the `prepare` and `write` phases. In a `prepare` phase it first compares the message's logical timestamp with its own. If it is greater the server updates its current time and sends an acknowledgement; otherwise it ignores the request and leaves its state unchanged. For a `write` request, if the message's timestamp equals the server's then the server computes the new value with the update function and overwrites its state with the message's. In the final case a client begins a `prepare` phase by incrementing its local time, resetting its snapshot, and broadcasting a `prepare` request to the servers.

Client and Server Specifications. Whereas `interpNet` defines how to handle network events, the functions in Fig. 19 show how clients and servers produce them. Clients can executing either `prepare` or `write`, both of which follow a common pattern (`sendReq`). First a `Begin` ghost event indicates that a new phase is starting and the state machine returns the message to broadcast to the servers. The server responses are then collected with `recvAll`. Finally, an `End` ghost event is

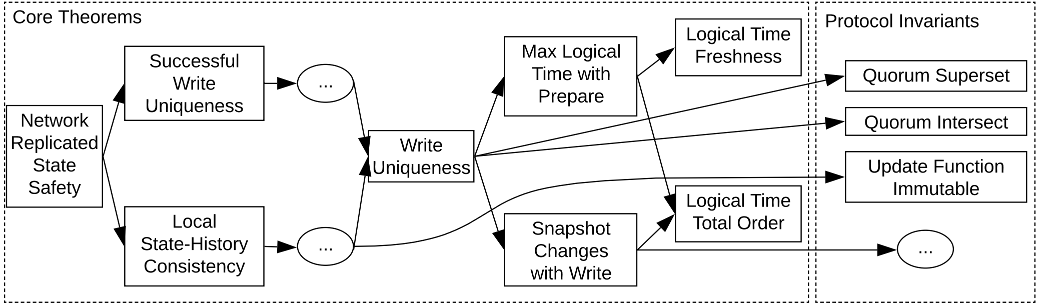


Fig. 20. Abridged proof dependencies.

added and the success of the operation is determined by using `is_q` to check if a quorum of servers acknowledged the request. Server behavior is described by `acqReq`, which simply waits for and responds to requests.

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 Network Replicated State Safety (Section 5.3), we have to show that all histories associated with a certain logical time are equal. This depends on two other core theorems.

Core Theorem 1 (Local State-History Consistency)**.** *For every value in a client's snapshot history there exists a corresponding successful* `write` *message in the network history.*

Here, a successful `write` message is a request to update state that was accepted by a quorum of servers. Core Theorem 1 relates each element in the `write` history to the corresponding network communication.

Core Theorem 2 (Successful Write Uniqueness)**.** *Any two successful* `write` *messages associated with the same logical time contain the same* `write` *history.*

Proof. We proceed by induction on the network history. Because logical times are unique (Core Lemma 5), the two `write` messages must come from the same client. Then, because an `write` message must follow a `prepare` phase, there are two cases: 1) a `prepare` occurred between the `write`, or 2) there is an intermediate third `write`. In the first case we are done because elections change the logical time (Core Lemma 3), which contradicts the assumption that they are equal. In the second case, we use the inductive hypothesis to show that the first and intermediate `write` messages send the same history and then again derive a contradiction from the change in snapshot length between the intermediate and the last `write`. □

Core Theorem 2 expresses a kind of immutability; once a `write` is committed in a particular position associated with a certain 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 [Rystsov 2018]), 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 (Write Uniqueness)**.** *No two* `write` *messages have the same logical time and snapshot history length.*

Core Lemma 2 (Snapshot Changes with Commit)**.** *A client's local value changes only at the end of a `prepare` phase (when it is copied from servers) or at the end of a `write` operation (when a new update is appended to it).*

Core Lemma 3 (Max Logical Time with Prepare)**.** *The latest logical time in the corresponding value after a successful `prepare` is the maximum among the participating nodes.*

Core Lemma 4 (Logical Time Total Order)**.** *Logical times are totally ordered.*

Core Lemma 5 (Logical Time Freshness)**.** *Incrementing the logical time creates a fresh value.*

Core Lemma 1 guarantees that either the time or snapshot has to change between `write` phases. Core Lemma 2 describes when and how a client's state changes. Core Lemmas 3 to 5 guarantee basic properties about the logical timestamps.

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.

The proof of Core Theorem 2 also critically depends on another low-level core theorem that expresses one of the defining properties of quorum-based replicated distributed systems.

Protocol Invariant 1 (Quorum Intersect)**.** *If $\mathbb{S}_1$ and $\mathbb{S}_2$ are quorums (with respect to `is_q`) for `prepare` and `write` phases in the same logical time respectively, then the two sets have a non-empty intersection.*

Protocol Invariant 2 (Quorum Superset)**.** *If $\mathbb{S}$ is a quorum, and $\mathbb{S} \subseteq \mathbb{S}'$ then $\mathbb{S}'$ is also a quorum.*

Protocol Invariant 3 (Update Function Immutable)**.** *Calling the update function (`update_func`) on a particular position in the log commits that value and later updates to the same position have no effect.*

Protocol Invariants 1 and 2 are defining properties of quorum-based consensus protocols and are straightforward to prove with set-theoretic arguments for most reasonable implementations of `is_q`. Protocol Invariant 3 enforces the write-once property of Paxos logs and holds trivially for most update functions.