Adore: Atomic Distributed Objects with Certified Reconfiguration

Wolf Honoré  
Yale University  
New Haven, Connecticut, USA  
wolf.honore@yale.edu

Ji-Yong Shin  
Northeastern University  
Boston, Massachusetts, USA  
j.shin@northeastern.edu

Jieung Kim∗  
Yale University  
New Haven, Connecticut, USA  
jieungkim@google.com

Zhong Shao  
Yale University  
New Haven, Connecticut, USA  
zhong.shao@yale.edu

Abstract
Finding the right abstraction is critical for reasoning about complex systems such as distributed protocols like Paxos and Raft. Despite a recent abundance of impressive verification work in this area, we claim the ways that past efforts model distributed state are not ideal for protocol-level reasoning: they either hide important details, or leak too much complexity from the network. As evidence we observe that nearly all of them avoid the complex, but important issue of reconfiguration. Reconfiguration’s primary challenge lies in how it interacts with a protocol’s core safety invariants. To handle this increased complexity, we introduce the Adore model, whose novel abstract state hides network-level communications while capturing dependencies between committed and uncommitted states, as well as metadata like election quorums. It includes first-class support for a generic reconfiguration command that can be instantiated with a variety of implementations. Under this model, the subtle interactions between reconfiguration and the core protocol become clear, and with this insight we completed the first mechanized proof of safety of a reconfigurable consensus protocol.

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

Keywords: distributed systems, consensus protocols, reconfiguration, formal verification, refinement, proof assistants

∗Jieung Kim is now at Google Research.

1 Introduction
When reasoning about distributed systems there are three main layers of abstraction to consider. On top is an application, which provides some client-facing interface for a replicated object (e.g., Chubby [1] or ZooKeeper [11]). Under that is a distributed protocol like Paxos [12, 29] or Raft [26], which manages replication and consistency. At the bottom is the network level, which implements the communication primitives used by the protocol and handles issues such as server crashes and network asynchrony.

Each layer has different goals and challenges, so naturally it is important to model the state of a distributed system in a way that suits the properties being proved. Previous work has developed useful state abstractions at various levels [7, 20, 27, 28, 38]; however, we claim that none has yet found the right model for protocol-level reasoning. This is especially true if one considers a critical feature that, thus far, has been largely unaddressed by verification work: reconfiguration.

Reconfiguration is necessary for realistic distributed systems, but it deeply interacts with a protocol’s core invariants in a way that makes it difficult to verify. The ideal protocol-level model should therefore express these invariants as clearly as possible while also hiding all irrelevant details from the network and application levels. It should also be general enough to support many different protocols and reconfiguration schemes so that proofs can be reused. In this paper we present Adore, a novel and generic approach to modeling reconfigurable consensus protocols that is designed for protocol-level reasoning.

There is no clear divide between different abstraction layers, but rather a continuous spectrum. Fig. 1 shows some of the most commonly used models and how Adore fits into

ACM Reference Format:  
their hierarchy (these are explained in more detail in Section 2.2). At one end is the class of network-based models, which represent a protocol’s behavior in terms of a set of servers passing messages over an abstract network. This is flexible and closely mirrors the implementation; however, it blends protocol and network-level logic, which obscures the protocol safety invariants and creates a huge number of cases to consider. If the set of participating servers is then also allowed to change, the situation quickly becomes untenable.

On the other end is state machine replication (SMR), which gives the illusion of a single, atomically-accessible object (represented by a log of committed commands) rather than a collection of individual servers. It hides internal communication details and the existence of intermediate, inconsistent states behind a remote procedure call (RPC) interface. This makes it ideal for application-level reasoning, but much too abstract to prove anything about a protocol.

A third option just below SMR is the atomic distributed object (ADO) model [8], which also provides an atomic object-oriented interface, but “opens up” the black box slightly. It unfolds the RPC call into three atomic steps, each of which can fail, and, in addition to committed commands, it preserves and exposes intermediate states with a tree of uncommitted commands. This is necessary to accurately model partial failures, which affect both protocol and application-level behaviors. However, the ADO model is mainly targeted at application-level reasoning and hides too many details to be appropriate for use at the protocol level.

Adore builds on the ADO’s core concepts and adds the missing details that allow it to represent a protocol’s entire history including committed states, partial failures, and configuration changes in a single tree. This makes invariants and dependencies of the replicated state clearer than when it is scattered across different servers. Communication is abstracted into an atomic ADO-like interface, which hides many of the network-level complexities.

Adore’s support for reconfiguration is especially important because server failures are inevitable in distributed settings [5, 23], so a method for safely and efficiently adjusting the membership is essential. In particular, Adore targets the safety of the challenging class of “hot” algorithms, which continue processing client requests during a membership change. Its generic reconfiguration command is parameterized by the definition of a quorum and of a valid configuration, which also allows it to admit a variety of implementations.

Reconfiguration requires special care because it can break many of the basic assumptions that protocols rely on. Adding or removing a server at the wrong time can easily compromise a protocol’s safety by allowing committed data to be overwritten, or liveness by making the entire system inoperable [6]. The fundamental challenge is that reconfiguration changes the metadata that protocols rely on to achieve consensus (e.g., membership, quorum sizes), but it also uses consensus to ensure the changes are applied consistently.

This circularity creates subtle dependencies among different aspects of the protocol, so it can be difficult even for experts to fully anticipate how reconfiguration influences the safety properties. For example, Raft’s single-server membership change algorithm contained a critical safety bug that passed unnoticed for over a year after its publication [24, 25]. A fix was proposed along with a loose sketch of its correctness, but up to now no complete proof has been published.

Using the Coq proof assistant [36] we prove that Adore satisfies the key safety property that committed states are never lost or overwritten. This guarantee applies to any benign fault tolerant consensus algorithm with a compatible hot reconfiguration scheme (defined in Section 3) that refines Adore. This proof brings to light subtle circularity issues that were not apparent in previous informal proof sketches, but we demonstrate that with Adore, an elegant solution naturally arises from its novel tree-based abstraction.

Our key contributions are:

- **Adore**: A novel protocol-level model, whose tree-based abstract state enables simple verification of the safety of reconfigurable distributed consensus protocols by hiding irrelevant network-level details and neatly capturing important state dependencies and invariants.
- A mechanized proof that the reconfigurable protocols modeled by Adore satisfy replicated state safety, which is the first of its kind. This proof holds for any reconfiguration scheme that satisfies the assumptions about the parameterized quorum and configuration definitions.
- Multiple case studies demonstrating the generality of our framework and covering reconfiguration schemes such as Raft single-node [24, 25], Raft joint consensus [26], primary backup [30], and dynamic quorum sizes [17].
• A connection to a more standard network-based specification of a Raft-like protocol through a refinement proof.
• Automated extraction from the Coq specification of the Raft-like protocol to an executable OCaml program.

The source code is available on Zenodo [10]. Additional details can be found in the appendices of the extended technical report [9].

2 Overview

Our goal is to design a language-based abstraction to facilitate verification of distributed protocols with reconfiguration. Before coming to our solution, we first provide some background on the protocols, ways in which existing models are lacking, and how reconfiguration complicates the problem.

2.1 Consensus

There are many types of distributed system, but we target the important class of consensus protocols. The high-level goal is to replicate a log of commands across n servers (or replicas) and get them to agree on the command in each slot. This is driven by a leader replica, which commits commands by convincing a large-enough subset (often a majority) of the replicas, known as a quorum, to add them to their logs.

Two popular consensus protocols are multi-Paxos [29] and Raft [26]. Each has an election phase in which replicas vote for a leader with a sufficiently up-to-date log (determined by comparing the logical timestamp of the last entry), followed by a commit phase where the leader replicates new commands. See the appendices for a more detailed tutorial.

Consensus guarantees replicated state safety, which ensures global agreement on the order of committed commands. This holds even if some replicas crash, as long as a quorum remains functional. This is because election and commit phases both require quorums, which prevents replicas from being partitioned into isolated groups since every action must be approved by at least one member of both groups.

2.2 Distributed System Models

Finding the right model that highlights key properties while hiding unnecessary details can reveal elegant solutions. For consensus this means cleanly representing the implicit relation between different replicas’ local states; i.e., the existence of a common prefix of committed commands.

2.2.1 State Machine Replication. State machine replication (SMR) [31] is a popular high-level abstraction that models the distributed state as a centralized, global log of commands. Clients extend the log through an opaque remote procedure call (RPC) [1, 39] interface, which internally relies on an underlying consensus protocol.

For example, consider a distributed key-value store with a put command to insert a new mapping. From a client’s perspective, put("a", 1) is an atomic action that either updates the state, or times out and fails (SMR in Fig. 2). Internally, however, a replica may initiate an election and repeatedly multicast the command to handle partial failures.

This is a convenient abstraction for clients of a distributed application who are not concerned with its inner workings, but it hides too much information about quorums and uncommitted commands to be suitable for protocol-level reasoning.

2.2.2 Network-Based Models. One can recover these details with a lower-level network-based model, like those used in many existing verification projects [7, 34, 35, 38]. This mirrors the implementation and models the system as a set of logs (one per replica). Rather than an atomic RPC interface, communication is modeled with abstract network events (e.g., send an election request, receive a vote). In this case put("a", 1) is no longer an atomic operation, but a long sequence of interleaving events (Network in Fig. 2).

It is clearly possible, though challenging, to prove a protocol’s correctness in this type of model as evidenced by proofs for Raft [40] and multi-Paxos [7]; however, the model does little to highlight protocol-level invariants and it allows network-level implementation details to leak through. For example, the representation of the local states as independent logs, while true to the implementation, does not make it clear that they must agree on a prefix of committed entries.

2.2.3 ADO Model. A third option is the atomic distributed object (ADO) model [8], which provides an atomic interface and a centralized, tree-based state abstraction, while still exposing distinct election and commit phases as well as uncommitted states. Like SMR, it is targeted more at application rather than protocol-level verification; nonetheless, some of its core design decisions are a step in the right direction.

State. Like SMR, committed methods in the ADO model are represented by an append-only persistent log. Uncommitted methods, which SMR does not explicitly model, are
referred to as caches (due to their volatile nature) and are organized into a cache tree. The parent-child relation represents dependencies between methods that must be respected if they are ever committed and "flushed" to the persistent log. For example, if P is the parent of C, then either P or the sequence P • C can be committed, but never only C. Forks in the tree represent slots for which replicas’ logs disagree.

Fig. 1 shows a side-by-side comparison of a set of multi-Paxos or Raft logs at the bottom (i.e., network-based) and the equivalent ADO persistent log and cache tree on the second row. A majority of replicas at the bottom agree on M1 and M2, so they are committed. This is reflected in the ADO model’s persistent log by the squares. S1 and S2 have M1 and M3 in the third slot, respectively. Since one replica is not a quorum in a three- replica configuration, both M3 and M5 are uncommitted, hence the circles in the ADO cache tree. As S1 and S2 disagree about the methods in this third slot, M4 and M5 are represented in different branches in the tree. M4 is uncommitted and comes after M3 in S1, so it is placed on the same branch as M3 in the ADO cache tree.

The persistent log and cache tree provide a neat, global summary of the replicas’ local states. The interface for atomically interacting with them consists of invoking application-defined methods, and the pull and push operations, which represent the election and commit phases, respectively.

Pull. Like the election phase, the purpose of pull is to choose a unique timestamp and an up-to-date state snapshot. Rather than broadcast a request and wait for a quorum of responses, the ADO model assumes the existence of an oracle that abstracts over the network communication to nondeterministically return either a successful or failed outcome.

On success the oracle chooses a unique timestamp and an up-to-date state snapshot. The interface for atomically interacting with them consists of invoking application-defined methods, and the pull and push operations, which represent the election and commit phases, respectively.

Method Invocation. Method invocation adds a child to the caller’s active cache, and sets the new cache as active (e.g., S1 and M6 in Fig. 1c). Only leaders are allowed to invoke methods, but multiple leaders can be active at the same time (e.g., S1 and S2 in Figs. 1d and 1e). This is safe because only the one with the largest timestamp can commit its methods.

Push. The push operation attempts to commit the methods on the leader’s active branch (the ancestors of the active cache). Like pull, an oracle nondeterministically decides the outcome. When committing multiple methods it may happen that some succeed while others are lost due to network errors, but if one fails then all that follow it must as well.

The oracle models this outcome by selecting an arbitrary prefix of the active branch to commit. The successful prefix is moved to the persistent log while the uncommitted suffix remains in the tree. The sibling branches now represent stale states so they are also removed. In Fig. 1f, S2 only manages to commit M1 and M5, leaving M2 in the cache tree.

Returning to the key-value store, calling put("a", 1) is a hybrid of the steps in SMR and network-based models (ADO in Fig. 2). If the client does not yet have an active cache it first calls pull. It then invokes the method and tries to commit it with push. Each step may fail, in which case the client decides to retry or abandon the attempt. This unfolds the RPC interface like the network-based approach, except each step is atomic, and the state is collected into a centralized tree-based representation rather than split into local logs.

2.3 Reconfiguration

In practical systems the set of participating replicas may not be constant as old servers are taken down for maintenance and new ones are added to cope with increased load. Because the key to maintaining safety is that elections and commits have overlapping quorums, the configuration change must be handled carefully to avoid violating this invariant.

There are many algorithms to accomplish this [16]. Some, such as Stoppable Paxos [21], use a “stop-the-world” approach that first blocks new commands from being committed by the old configuration, then copies the logs to the new configuration, and finally resumes normal processing. This somewhat simplifies the problem by ensuring that at no point are both configurations active at once; however, it also incurs a performance cost due to the disruption in service.

An alternative is “hot” reconfiguration, which alters the configuration without blocking the normal processing of commands. Although clearly a more attractive option, this introduces additional complexities by intermingling the old
and new configurations. Adore supports many hot algorithms, but to highlight the main challenges we first consider Raft’s single-node membership change approach [24].

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

- **R1** A new configuration can differ from the leader’s configuration by at most one server.
- **R2** The leader’s log cannot contain any uncommitted reconfiguration commands.

These restrictions are meant to ensure that consecutive configurations still have overlapping quorums, so the usual (static configuration) safety arguments still hold. R1 guarantees that a majority subset of a new configuration still shares at least one server with the old one, and R2 makes sure configurations only change once before being committed. At first glance these seem sufficient; however, they miss a subtle, but critical corner case that took over a year to discover [25].

**The Problem.** Consider Fig. 4, in which the configuration is $S_1$–$S_4$ and $S_1$ is the leader. $S_1$ proposes a new configuration without $S_4$, but fails to replicate it. $S_2$ then initiates an election, becomes the leader with the support of $S_3$ and $S_4$, and begins its own reconfiguration that removes $S_3$. $S_2$ begins using its new configuration immediately, so the command is committed once it reaches $S_4$ ($\{S_2, S_4\}$ is a majority of $\{S_1, S_2, S_3\}$). Suppose then $S_1$ initiates another election and receives votes from itself and $S_1$. Because it also uses the latest configuration from its log ($\{S_1, S_2, S_3\}$), these two votes constitute a quorum and it wins the election. At this point the game is lost because $S_1$ and $S_2$ are leaders with disjoint quorums, which means each is free to commit commands independently, violating the consistency guarantee.

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

**The Solution.** The solution proposed by Ongaro [25] is to deny pending reconfiguration commands before issuing new ones. Because a leader can only be elected if its log contains every committed command, it is enough to commit any regular command before allowing reconfiguration.

- **R3** The leader’s log must contain a committed command with the current timestamp.

Ongaro [25] gives a very high-level proof sketch that R3 solves the problem by arguing that a leader cannot be elected without the latest committed command in its log. It enumerates the possible configurations for the leader and command and shows that in each case their quorums must overlap because of R1–3. As before, this argument seems reasonable, but it overlooks some subtle issues. It relies on the invariant that leaders are elected with unique timestamps, but this requires some care since the leaders’ quorums may no longer overlap. Section 4 shows how the wrong approach can lead to a circular argument, and why a model that cleanly exposes reconfiguration’s mutual relation with consensus is needed.

### 2.4 Adore

The ADO’s cache tree helps expose the implicit dependencies and relationships between methods in the local logs, but it lacks metadata such as who the current leader is, and who voted for it. The key insight of Adore is that this information can be encoded in the cache tree, which makes it not just a representation of the current state, but also a complete history of how the system arrived at that state.

**Cache Tree.** To make this possible, in addition to recording the methods that have been called, Adore’s has new cache variants for metadata about leader elections and committed methods (ECaches for elections, CCaches for commits, and MCaches for methods). Every cache is also annotated with its configuration and a set of supporters; i.e., the replicas that voted for it. Finally, to support reconfiguration an
RCache is also introduced that behaves like an MCACHE, but contains a new configuration instead of a method.

An advantage of this representation is it gives a concise visual summary of the complete history of the distributed state. Fig. 5 shows the evolution of a system through a sequence of operations similar to Fig. 3. Invoking a method creates a leaf MCACHE node on whatever branch the caller was last active (Fig. 5b). Similarly, push places a CCACHE after one of the caller’s latest MCACHES, though not necessarily the last one (Fig. 5c). Like invoking a method, a reconfiguration grows the caller’s active branch (Fig. 5d). An ECACHE is inserted after the most up-to-date cache observed by any of the election voters (Fig. 5e). Here that is the CCACHE, because neither of the supporters, S2 and S1, have observed S1’s MCACHE or RCache yet. Note that unlike Fig. 1 where committed methods are represented by squares, MCACHES and RCache are always drawn as circles, and are implicitly committed if a CCACHE is among their descendants. This allows the cache tree to be append-only and avoid in-place updates.

By encoding the information this way, previously implicit relations on disjoint states are now more explicitly expressed as structural invariants. This also makes it easy to see the mutual dependency between RCache, ECache, and CCACHE that is the source of much of the safety proof’s complexity. Thanks to this insight we discovered a technique for breaking the circularity by decomposing the tree into subtrees with only one RCache each and then composing these more manageable cases to prove the safety of the general case.

Generality. The cache tree abstraction may seem far removed from a protocol’s actual implementation, but it is really the same information restructured to highlight the important details. We can prove this with refinement, which implies that Adore captures every valid behavior of a network-based model of a protocol, and therefore Adore’s safety properties hold for the protocol as well. Adore’s push, invoke, and reconfig operations map fairly directly onto the election, commit, and local log update phases found in most consensus protocols, so in fact, this relation can be proved for many protocols, including various Paxos variants and Raft. Section 5 demonstrates this in more detail.

The other dimension in which Adore is general is its reconfiguration scheme. Just as Raft’s single-node algorithm requires R1–3 to be safe, Adore has similar conditions that must be met. However, we observe that R1 is stronger than necessary and instead all that is needed is that consecutive configurations have overlapping quorums. In fact, the protocol’s safety is completely independent from the definitions of a quorum and of a valid configuration as long as they guarantee this property. By parameterizing these features Adore becomes a generic verification framework that permits many possible implementation.

Figure 6. Adore state definitions.

3 Adore Formal Semantics

This section formalizes the previous intuitive description of Adore. We mark everything related to reconfiguration in blue with a box. Removing these parts leaves a configuration-aware model (CADO) that is also useful for reasoning about the safety of protocols with static configurations.

State. Fig. 6 defines the type \( \Sigma_{\text{Adore}} \) for state \((st)\), which is a pair of a cache tree \((tree)\), and the largest timestamp that each replica has observed \((times)\). We use the notation
Parameters

\begin{align*}
\text{Config} : & \text{Type} \\
\text{isQuorum} : & \text{Set}(\mathbb{N}_{\text{nid}}) \rightarrow \text{Config} \rightarrow \mathbb{B} \\
\text{mbrs} : & \text{Config} \rightarrow \text{Set}(\mathbb{N}_{\text{nid}}) \\
\text{R1}^*: & \text{Config} \rightarrow \text{Config} \rightarrow \mathbb{B}
\end{align*}

Assumptions about \(\text{R1}^*\) and \text{isQuorum}

\begin{align*}
(\text{REFLEXIVE}) \ & \text{R1}^*(\mathcal{C}, \mathcal{C}) \land \text{isQuorum}(\mathcal{C}) \land \text{isQuorum}(\mathcal{Q}, \mathcal{C}) \\
\implies & \mathcal{Q} \land \mathcal{Q}' \neq \emptyset
\end{align*}

Definitions

\begin{align*}
\text{R2}(t, \mathcal{C}) & \triangleq \forall \mathcal{C}' \in \mathcal{C} \cup \mathcal{C}' \Rightarrow \exists \mathcal{C}' \in \mathcal{C} \cup \mathcal{C}' \subseteq \mathcal{C} \land \mathcal{C}' \cup \mathcal{C}' \cup \mathcal{C} \\
\text{R3}(t, \mathcal{C}) & \triangleq \exists \mathcal{C}' \in \mathcal{C} \\
\text{canReconf}(t, \mathcal{C}, \mathcal{C}', \mathcal{C}'' \cup \mathcal{C}) & \triangleq \text{R2}(t, \mathcal{C}) \land \text{R3}(t, \mathcal{C}) \\
\text{canReconf}(t, \mathcal{C}, \mathcal{C}') & \triangleq \text{R2}(t, \mathcal{C}) \land \text{R3}(t, \mathcal{C})
\end{align*}

Figure 7. Configuration/quorum parameters and definitions.

\(\text{name(st)}\) to represent extracting one of these fields (e.g., \text{tree(st)} returns the first element). Fig. 7 declares that the type of the configuration (\text{Config}) is an opaque parameter with functions to extract a set of nodes (\text{mbrs}) and decide if some set constitutes a quorum (\text{isQuorum}). This allows the model and safety proof to work for any instantiation of these parameters as long as they satisfy the \text{REFLEXIVE} and \text{OVERLAP} invariants (\(\mathcal{C}' \cup \mathcal{C} \subseteq \mathcal{C}' \cup \mathcal{C} \)), which implies global agreement on a consistent commit history.

Note that, unlike the ADO model, Adore does not have an explicit log of committed methods separate from the cache tree. This is because the ADO model assumes the underlying protocol ensures consistency, so the existence of a common prefix of committed methods is guaranteed. In Adore, on the other hand, we prove this property by showing that all \text{CCache}s lie on the same branch of the cache tree, which implies global agreement on a consistent commit history.

\text{Caches}. Caches are divided into election (\text{ECache}), method (\text{MCache}), reconfiguration (\text{RCache}), and commit (\text{CCache}) variants. Each has a unique cache ID (\text{cid}) and the cache tree is implemented as a partial map from \text{cid} to the corresponding cache, plus the \text{cid} of the cache’s parent (with 0 reserved for the root). The functions for growing the tree are \text{addLeaf}, which adds a child to a parent, and \text{insertBtw}, which inserts a cache between a parent and its children. We omit their technical details for brevity. For the unabridged semantics, refer to the appendices or the source code [10].

Each type of cache contains the node ID (\text{nid}) of the replica that called the operation creating it (\text{caller}), a timestamp (\text{time}), a version number (\text{vsn}), and the configuration (\text{conf}) under which it was called (the root cache is initialized with some \text{conf}_0). The timestamp corresponds to a Paxos ballot number or a Raft term number and is assigned to a leader in each round. The version number resets to 0 at the start of each round and increments on every method/reconfig call.

\(\text{Op} \triangleq \text{pull} : \mathbb{N}_{\text{nid}} \rightarrow \Sigma_{\text{Adore}} \rightarrow \Sigma_{\text{Adore}} \\
\text{invoke} : \mathbb{N}_{\text{nid}} \rightarrow \text{Method} \rightarrow \Sigma_{\text{Adore}} \rightarrow \Sigma_{\text{Adore}} \\
\text{reconfig} : \mathbb{N}_{\text{nid}} \rightarrow \text{Config} \rightarrow \Sigma_{\text{Adore}} \rightarrow \Sigma_{\text{Adore}} \\
\text{push} : \mathbb{N}_{\text{nid}} \rightarrow \Sigma_{\text{Adore}} \rightarrow \Sigma_{\text{Adore}}
\)

Figure 8. Adore operations.

In Fig. 9, we define a strict order (\(\succ\)) on caches by comparing the lexicographic order of their timestamp and version number pairs, with the exception that if a \text{CCache} has the same timestamp and version as a non-\text{CCache}, the \text{CCache} is considered greater, which is needed to make \(\succ\) a total order. \text{ECache}s and \text{CCache}s also carry the node IDs of their \text{supporters}; i.e., the replicas that voted for them. An \text{MCache} or \text{RCache}’s only \text{supporter} is its caller. \text{MCache}s also contain the name of a \text{Method}. In practice this encodes an application-specific function (e.g., add n to a counter) to be applied once committed; however, the actual methods have no bearing on the protocol’s safety, so we treat them as arbitrary identifiers.

\text{Operations}. Each of Adore’s operations (\text{pull}, \text{invoke}, \text{reconfig}, and \text{push}) takes its caller’s node ID, the current state (\(\Sigma_{\text{Adore}}\)), and for \text{invoke} and \text{reconfig} a new method or configuration, and returns a new state (see Fig. 8). The \text{pull} and \text{push} operations rely on an oracle \(\odot\) (consisting of \(\odot_{\text{pull}}\) and \(\odot_{\text{push}}\) respectively) to model nondeterministic network behaviors. The semantics of each operation are defined in Fig. 10 (with helper functions defined in Fig. 9). We write \(\odot \triangleright op : st \leadsto st'\) to mean calling the operation \(op\) on state \(st\) with oracle \(\odot\) results in \(st'\).

\text{Pull}. Recall that the purpose of the election phase is to choose a unique logical timestamp and a sufficiently up-to-date state snapshot with all of the committed commands.
demonstrates a few of the many possible
raky in Fig. 10. This is also a possible outcome for the
other operations, which we omit in Fig. 10.

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

Reconfig. So far every new cache inherits its parent’s
configuration. The only exception is an RCache, which is
especially a special kind of MCache that contains a new
configuration (ncf) instead of a method. RCacheS are created
by reconfig, whose semantics are nearly identical to regular
method invocation, except for a few additional restrictions,
which are modified versions of R1–3 from Section 2.3. Put
in words R2 and R3 guarantee the following:

R2 There are no uncommitted RCacheS in the active branch.

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

As explained earlier, Raft’s R1 is stronger than necessary
so it is replaced by the more general R1∗ predicate, which
can be instantiated by any condition that satisfies the properties
in Fig. 7. Section 6 demonstrates a few of the many possible
reconfiguration schemes this permits.

Push. Like the ADORE model, a successful push commits an
arbitrary prefix of the most recent uncommitted commands.
This is chosen by Ωpush, which returns a cache (CM) that
satisfies canCommit. This means CM must be an MCache or
RCache that was called by nid with its current timestamp,
and is more recent than the latest CCache supported by nid.
This guarantees that nid is a valid leader, CM is an
uncommitted command, and committing it will not conflict with
a previous commit. Like pu1, the supporters’ timestamps
must not be greater than CM’s, though they may be equal.

As with pu1, push updates the supporters’ timestamps,
and the cache tree if Q is a quorum. The new CCache (Cnew)
is added with insertBtw rather than addLeaf, which puts
Cnew between CM and its children instead of creating a leaf
node. The children represent partial failures that may still be
committed later on, so shifting them after the CCache leaves
them as viable candidates for some later pu1 or push.

4 Safety Proof

The primary purpose of ADORE is to simplify the verifica-
tion of safety properties of consensus protocols even with
the complexity introduced by reconfiguration. This section
demonstrates how it accomplishes this goal by sketching
the proof of replicated state safety and highlighting inter-
esting challenges. For reasons of clarity and space, we stick
mainly to informal arguments, but more rigorous proofs can be found in both the appendices and the source code [10].

4.1 Breaking Circularity with rdist

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

**Definition 4.1** (Replicated State Safety). There exists a linear path from the root of the tree to a leaf node that contains every committed method. In other words, for any CCaches C_{C1} and C_{C2}, one must be a descendant of the other.

Without reconfiguration, the core of this proof is that consecutive elections and commits both require a quorum of supporters. This implies that they share at least one replica, which ensures that a leader’s log must be sufficiently up-to-date and contains the latest committed method. Note, however, that this latest commit is unique only if every committed method has a distinct timestamp and version number pair, which is easy to prove as long as every leader has a unique timestamp. The standard proof of this property reasons that leaders require a quorum of voters, so two elections must involve at least one common voter. Then, because replicas only vote for candidates with timestamps greater than what they have seen, the shared voter cannot have voted for two candidates with the same timestamp.

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

This circularity arises because there may be arbitrarily many reconfigurations between the commit and election. The solution is to count the number of reconfigurations between two commands, which we call their rdist, and reduce the problem to smaller, more manageable steps. This is a fairly awkward property to express in a network-based specification because one must essentially construct a tree from two logs by merging their common prefix into a branch that forks where their tails diverge. This is much more natural in Adore because the cache tree already captures this structure.

---

**Definition 4.2** (rdist). Suppose C_1 and C_2 are caches with a nearest common ancestor C_A. The rdist of C_1 and C_2 is the number of RCaches on the path between C_1 and C_2, which passes through C_A, not including the endpoints.

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

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

rdist = 1 R1*, R2, and R3 guarantee quorums still overlap. rdist = n + 1 The cache tree must decompose into a subtree with rdist = n and a branch with exactly one RCache. The inductive hypothesis and rdist = 1 case guarantee CCaches in these regions are on the same branch.

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

An advantage of Adore is that one can instead reason directly about the structure of the cache tree, which makes for simpler and more intuitive proofs. Given a cache tree, one can consider a small number of cases that could have led to that situation, and prove that the property holds for each. These cases are often most easily explained using pictures like the one to the left. This represents a subtree in which Z is a common ancestor of X and Y. The cloud symbol means Z can be any type of cache. The dotted arrows indicate that X and Y are descendants, but not necessarily direct children of Z. The label rdist = n means that rdist(X, Y) = n.

4.2 Base Cases

The safety proof for the rdist = 0 case follows the standard static-configuration argument [14, 26] so we leave the details to the appendices. Many properties that hold for caches where rdist = 0 also hold when rdist = 1 if one can show that their configurations still have overlapping quorums, which is precisely what R1*, R2, and R3 are meant to guarantee. The purpose of R1* is clear (OVERLAP in Fig. 7 ensures that when a leader proposes a new configuration it overlaps with the old one), but the other two are no less important.

R2 ensures that a leader cannot begin a new reconfiguration attempt while there is an uncommitted RCache in its branch. This prevents the configuration from changing twice in a single commit, which might break the overlap guarantee (OVERLAP only holds for consecutive configurations). R3 requires the leader’s log to contain a committed entry with the current timestamp. This serves a similar purpose to R2.
implies but with cache trees). The leader (this is the \(\text{safety, } \text{rdist} \leq r \))

ticipate in \(\text{rdist} \) from similar arguments to their \(\text{(configuration on a different branch from the reconfiguration, which is compromised. However, because } S_1 \text{ and } S_2 \text{ did not participate in } S_2 \text{'s reconfiguration, } S_1 \text{ is elected using its own configuration on a different branch from the } \text{CCache.}

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

Together these properties guarantee that caches with an \(\text{rdist} \) of 1 have overlapping quorums and therefore properties like the uniqueness of a leader's timestamp and safety follow from similar arguments to their \(\text{rdist} = 0 \) counterparts.

**Theorem 4.3** (Safety, \(\text{rdist} \leq 1 \)). *Any cache tree } tr \text{ with } \text{rdist} \leq 1 \text{ satisfies replicated state safety.}

**4.3 General Case**

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

**Lemma 4.4** (CCache in \(\text{RCache} \) Fork). \(\text{Let } C_{R1} \text{ and } C_{R2} \text{ be } \text{RCaches} \) such that \(\text{rdist}(C_{R1}, C_{R2}) = 0 \), and neither is a descendant of the other, but both have a common ancestor \(C_A\). Then there exists a \(\text{CCache} \) \(C_C \) that is a descendant of \(C_A \) and an ancestor of either \(C_{R1} \) or \(C_{R2} \).

**Theorem 4.5** (Safety). *Any cache tree } tr \text{ with any } \text{rdist} \text{ satisfies replicated state safety.}

**Proof Sketch.** We proceed by induction on \(\text{rdist}(tr) \). For \(\text{rdist} \leq 1 \) we are done by Theorem 4.3. Suppose now that all trees with \(\text{rdist} = n \) are safe, and \(1 < \text{rdist}(tr) = n + 1 \) so 1 < \(\text{rdist}(C_{C1}, C_{C2}) \leq n + 1 \) for some \(\text{CCaches} \) \(C_{C1} \) and \(C_{C2} \). If \(\text{rdist}(C_{C1}, C_{C2}) \leq n \) then they are in some subtree \(tr' \) with \(\text{rdist}(tr') = n \), so we are done by the inductive hypothesis. Safety also holds if \(C_{C1} \) and \(C_{C2} \) are on the same branch, and if not we will show that all other shapes for } \text{tr} \text{ are impossible.

The first two cases are symmetric, and Lemma 4.4 implies that in the last case there must be another \(\text{CCache} \) between \(C_A \) and either \(C_{R1} \) or \(C_{R2} \), which results in the same situation as the other cases. Therefore, we can assume without loss of generality that \(C_R \) is on the first \(\text{RCache} \) on \(C_{C1}' \)s branch. Let \(C_{CR} \) be \(C_R \)'s first \(\text{CCache} \) descendant. It is enough to show that \(\text{rdist}(C_{CR}, C_{C2}) \leq n \) because then \(C_{CR} \) and \(C_{C2} \) must be on the same branch, which is a contradiction. We know \(\text{rdist}(C_{C1}, C_{C2}) > 1 \), so \(C_R \) cannot be the only \(\text{RCache} \) on \(C_{C1}' \)s branch. We also know by R2 that this other \(\text{RCache} \) cannot be between \(C_R \) and \(C_{CR} \). Therefore this \(\text{RCache} \) does not count towards \(\text{rdist}(C_{CR}, C_{C2}) \) and it is at most \(n \).

5 Refinement

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

In a sense, \(\text{Adore} \) models an abstract, synchronous version of Raft in which communication happens atomically and in logical time order. Through a series of refinements we show that a more realistic asynchronous network-based
specification of Raft behaves equivalently to this synchronous version. This part is similar to previous work \[2, 7, 37\], but an important distinction is the final refinement, which lifts the simplified network-based model to Adore.

**Specification.** We begin by writing a standard asynchronous network-level specification for our Raft protocol. The state (Fig. 13) comprises a set of replicas each with a current timestamp, a local log of commands, and some additional bookkeeping details (e.g., the current leader, number of votes received). Communication between replicas is only possible through the network, which is represented as a pair of bags of sent and delivered messages, respectively. Messages are of four types: election/commit requests/acknowledgements. Requests are generated by the elect and commit operations. Any time after being sent, a message may arrive with deliver, which triggers a handler based on the type of the message. A leader may also call invoke and reconfg, which are local operations that only affect its own log.

We also create another specification that is the same except for a few simplifying assumptions: only valid messages are delivered (i.e., messages that will not be ignored by their

recipients for having the wrong timestamp, coming from outside the current configuration, etc.), messages are delivered in order of their logical timestamps, requests are received and acknowledged atomically by all of the recipients at once. Unless otherwise qualified, we refer to the asynchronous model as Raft, and this simplified version as SRaft.

**Refinement Relation.** The next step is to define a refinement relation \( R \) between Raft and Adore’s state. This includes correspondences between auxiliary state such as timestamps, and leadership flags, but for Adore’s safety to imply something useful about Raft, \( R \) must ensure that for any replica’s local log, the methods are the same as those along that replica’s active branch of the cache tree. Then, since Adore guarantees that every replica’s active branch has a common prefix of committed methods, the same must be true in Raft as long as \( R \) holds. SRaft and Raft share the same state definitions, so \( R \) also holds for SRaft and Adore.

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

Intuitively, the correspondence between Adore and Raft steps is clear: pull for elections, push for commits, and Adore method invocation and reconfguration for their Raft counterparts. The reality is less straightforward because the Adore operations are atomic, while their Raft equivalents are not. SRaft’s purpose is to be an intermediate specification
that rearranges asynchronous Raft operations into an equivalent order that satisfies the intuitive mapping. As an example, consider the situation in Fig. 14. We’d like to show that the four Raft receive events in the bottom layer correspond to the two pull requests in the top layer, but how can we be sure that these sequences of events are actually equivalent?

We first observe that $S_1$ receives a message with timestamp 1 after it received one with timestamp 2. Therefore, $S_1$ ignores the second message, and we can safely drop it from the sequence of events. Next, we note that $S_1$ receiving a message has no effect on $S_2$ and vice-versa, so the first two receive events safely commute, which puts the sequence in logical time order. Now that $L_1$ and $L_2$’s requests are untangled, we can merge adjacent receive events and treat the messages as if they arrived at each recipient at the same time.

Now we have a much simpler network-based model with in-order, atomic message delivery that is equivalent to the asynchronous version. To complete the refinement, the final step is to show that corresponding SRaft and Adore operations preserve $R$. Because we have already “lined up” the events, the bulk of the remaining work is to translate between different state representations, such as a replica’s local log in SRaft and its active branch in Adore (Fig. 15). See the appendices for more information about these refinement layers, and the source code [10] for the full proofs.

6 Instantiating Reconfiguration Schemes

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

Raft Single-Node. One option, of course, is Raft’s single-node algorithm, which uses a standard majority quorum and only allows configurations to change one replica at a time.

\[
\text{Config} \triangleq \text{Set}(\mathbb{N}_{\text{mid}}) \times \text{Option}(\text{Set}(\mathbb{N}_{\text{mid}}))
\]

\[
R^+(C, C') \triangleq \exists \text{old}. (C = (\text{old}, \bot) \land C' = (\text{old}, _)) \lor \exists \text{new}. (C = (\_, \text{new}) \land C' = (\text{new}, \_))
\]

\[
isQuorum(S, (\text{old}, \text{new})) \triangleq (\text{old}) < 2 * (|S \land \text{old}|) \land (\text{new} = \bot \lor |\text{new}| < 2 * (|S \land \text{new}|))
\]

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

Primary Backup. Instead of relying on majorities, another approach is something similar to a primary backup protocol, such as Chain Replication [30], in which one replica or set of replicas (the primary) is responsible for committing commands, while the others serve as passive backups. A quorum then is any set containing the primary, which means the set of passive backups can change arbitrarily.

\[
\text{Config} \triangleq \mathbb{N}_{\text{mid}} \times \text{Set}(\mathbb{N}_{\text{mid}})
\]

\[
R^+((P, \_), (P', _)) \triangleq P = P'
\]

\[
isQuorum(S, (P, \_)) \triangleq P \in S
\]

In this case the primary is constant and is a member of every quorum, so all quorums obviously intersect. The limitation of this is if the primary crashes then all progress is blocked. A more reliable alternative is to use one of the previous approaches to manage a set of primaries that can be replaced as needed. Primaries can then be replaced one at a time, and passive backups can still be freely added or removed. This allows replicas to move between the primary and passive sets to dynamically adjust to varying availability needs.

Dynamic Quorum Sizes. With a set of $n$ replicas, a quorum size of $\lceil n/2 \rceil$ allows only one replica to be added or removed while still guaranteeing overlap. On the other hand, a quorum size of $n$ means $n - 1$ replicas can safely be changed at once. Larger quorums therefore allow for faster reconfiguration, but are less fault tolerant. This type of trade-off is why protocols like Vertical Paxos [17] allow quorum size to be adjusted dynamically. Adore’s reconfiguration scheme supports this by adding quorum size $(q)$ to the configuration.

\[
\text{Config} \triangleq \mathbb{N} \times \text{Set}(\mathbb{N}_{\text{mid}})
\]

\[
R^+((q, C), (q', C')) \triangleq (C \subseteq C' \lor |C'| < q + q') \lor (C' \subseteq C \lor |C| < q + q')
\]

\[
isQuorum(S, (q, C)) \triangleq q \leq |S \land C|
\]

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

Proof Effort and Experience. The total amount of Coq to implement and prove the safety of Adore is approximately 10.8k lines. Of that, 2.3k are generic well-formedness invariants about the tree data structure (e.g., proving the absence of cycles), and 4k are part of a general library of utility functions and lemmas, leaving 4.5k for the kind of proof shown in Section 4. We also performed the same safety proof using the CADO model (Adore without reconfiguration), which took approximately 1.3k lines (excluding the tree properties). The CADO proof took one person approximately 2 weeks to complete and adding reconfiguration took another 3.

For comparison, Advert (an ADO-based Coq verification framework) proved a similar replicated state safety property for a network-based model of a non-reconfigurable multi-Paxos in approximately 5k lines [8]. The relative ease with which Adore handles a much more complicated problem is a strong demonstration that an abstraction designed specifically for protocol-level reasoning is a powerful tool.

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

Refinement. The refinement proof between the network-based model of a Raft-like protocol and Adore takes approximately 13.8k lines of Coq, of which 2.5k is the refinement between SRaft and Adore. The protocol is parameterized by the same isQuorum and R1 predicates as Adore, which means the refinement proof actually holds for a large family of protocols with different reconfiguration schemes. Instantiating these parameters and proving that they satisfy the necessary properties is trivial. Adore’s codebase includes six examples (the four from Section 6 and two others) that take about 200 lines in total for both the definitions and proofs (several rely on the proof that majority subsets overlap, which is an additional 100 lines).

OCaml Extraction and Performance. Using Coq’s support for extraction to OCaml we created an executable version of the Raft network-based specification and with a small, unverified network library wrapper, evaluated its performance on Amazon EC2 with m4.xlarge instances. The experiment reconfigures after every 1000 client requests, starting with five nodes, dropping to three, then increasing back to five. Fig. 16 shows the maximum, mean, and minimum latencies for processing each client command over eight runs. Reconfiguration adds a small delay, especially when the number of nodes increases, but it is within the normal range of sporadic latency spikes. Our aim is not to make any strong performance claims, but merely to demonstrate that Adore’s safety guarantees can extend to verified executable code (excluding the OCaml extraction process, compiler, and network libraries) without being unreasonably slow.

8 Related Work

Alternative Reconfiguration Algorithms. Adore is designed to support hot reconfiguration algorithms where uncommitted reconfiguration commands update the configuration immediately. These are both more efficient and more challenging to verify than other types because they interleave reconfiguration with normal operations. The cache tree’s representation of uncommitted state is ideal for handling this kind of speculative behavior, and the R1+ and isQuorum parameters allow Adore to support a wide variety of schemes; however, we believe that with some slight modifications, it could easily handle even more.

Lamport et al. [16] suggests an “easy” approach to reconfiguration in which each instance of consensus (each slot in the log) uses a configuration that is inherited from the previous instance. The configuration can then be changed by committing a special command that tells the next consensus instance to use it. To avoid blocking instance $i + 1$ from beginning until $i$ is fully committed, the algorithm is generalized to use a parameter $\alpha$ such that a configuration committed in instance $i$ takes effect in instance $i + \alpha$, thus allowing $i$ through $i + \alpha - 1$ to continue as normal.

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

Stoppable Paxos [21] introduces a “stop” command that prevents replicas from committing further commands. Once stopped, a new configuration is launched and the old log is copied up to the stop command. WormSpace [35] implements a similar approach by “sealing” the configuration, which causes servers to reject subsequent requests, before starting a new instance. Liskov and Cowling [18] also define a membership change algorithm for Viewstamped Replication.
in which a special command defines the new configuration and begins a view change (i.e., election). Like the other approaches, handling of client requests is paused until the logs are completely transferred to the new configuration.

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

Formal Verification. There is surprisingly little prior work on formal verification of reconfiguration. Verdi’s proof of Raft’s safety does not consider either the single-node or joint consensus algorithms [38, 40]. IronRSL’s Paxos-based IronFleet also omits reconfiguration though they claim it “only requires additional developer time” [7]. Padon et al. [27] prove the safety of Vertical Paxos, but assume the existence of a correct external reconfiguration service, thus sidestepping the issue. Even in the blockchain world, where membership tends to be very flexible, verification efforts often make strict assumptions about configurations. For example, a proof of safety for the Stellar Consensus Protocol [22] assumes “arbitrary, but fixed configurations” [19].

The nearest work to Adore is the verification of MongoDB’s reconfiguration scheme [32, 33], which occurred concurrently with Adore’s development. Like Adore, the protocol is based on Raft’s single-node algorithm, but with an optimization that reconfiguration operations are stored separately from regular commands, which somewhat relaxes the dependencies between them, and means replicas only need to keep the latest configuration. However, there are several significant differences in our approaches and results.

The MongoDB work is specified in TLA* [13, 15] in a very abstract network-based model and its safety is proved with the TLA* proof system [3]. The specification is at a comparable level of abstraction to our Sraft specification where communication details are mostly hidden, but state is still modeled as local logs rather than a global cache tree. Unlike Adore, there is no refinement with a more realistic model, or the ability to extract executable code. The MongoDB specification is also for a fixed reconfiguration scheme, and lacks the generality of Adore’s isQuorum and R1* parameters.

The high-level structure of the safety proofs are quite different as well. Because MongoDB also uses a hot reconfiguration algorithm, it faces the same sort of circularity problems described in Section 4; however, without Adore’s tree structure to suggest the $rdist$-based approach, they resort to the more standard technique of establishing an inductive invariant that implies safety and is preserved by every step of the specification. This invariant must contain enough information both to prove safety and its own invariance, which means several mutually dependent properties must be bundled together. In particular, the MongoDB invariant is a conjunction of 20 high-level properties ranging from important safety guarantees like election safety and log matching to implementation details such as the uniqueness of a configuration’s term and version number.

With all of these invariants packed together the intuition behind why the protocol works is obscured, and the proof is more complex because it is harder to break down into smaller steps. Discovering the correct invariant alone took between 1–2 person-months using a counterexample-driven approach with a tool that attempts to detect invalid invariants. Actually proving that the invariant is inductive and implies safety took another 4 person-months. When compared with the 5 person-weeks to prove Adore’s safety, this supports our claim that finding the correct protocol-level abstraction is essential for scaling verification to more realistic and complex systems.

9 Conclusion and Future Work

Existing models for distributed systems are not ideal for protocol-level verification, which makes it difficult to reason about important, but complex operations like reconfiguration. We presented Adore, whose cache tree abstraction and atomic interface make it the right choice for this type of problem. We demonstrate this by showing how it facilitates the first mechanized safety proof for a generic consensus protocol with a hot reconfiguration algorithm.

Although Adore guarantees the safety of the protocols it models, it makes no claims about their liveness or availability. These important properties can also be compromised by an incorrect reconfiguration scheme, so they are natural targets for future extensions of Adore. This requires introducing a notion of time and an assumption of a partially synchronous network, but we expect that Adore’s intuitive tree-based state representation would be an asset here as well.

Another direction that Adore might be extended is to support byzantine fault tolerant protocols such as HotStuff [41] or Jolteon [4]. These use larger quorum sizes and additional machinery to handle malicious replicas, but their safety ultimately still relies on a logical tree of commands with overlapping quorums to prevent branching. Thus, we expect an Adore-like model would also work quite well in this domain.

Acknowledgments

We would like to thank our shepherd Tej Chajed and anonymous reviewers for their helpful feedback. This material is based upon work supported in part by NSF grants 2019285, 1945541, and 2118851, and by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-21-C-4018. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, DARPA, and NIWC Pacific.
References


[25] Diego Ongaro. 2015. bug in single-server membership changes. https://groups.google.com/g/raft-dev/c/t4xj6JTPe/m/d2D9lWrZa8J


393


