

Adore: Atomic Distributed Objects with Certified Reconfiguration

Wolf Honoré

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

Jieung Kim*

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

Ji-Yong Shin

Northeastern University
Boston, Massachusetts, USA
j.shin@northeastern.edu

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.

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).

PLDI '22, June 13–17, 2022, San Diego, CA, USA

© 2022 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-9265-5/22/06.

<https://doi.org/10.1145/3519939.3523444>

ACM Reference Format:

Wolf Honoré, Ji-Yong Shin, Jieung Kim, and Zhong Shao. 2022. Adore: Atomic Distributed Objects with Certified Reconfiguration. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22)*, June 13–17, 2022, San Diego, CA, USA. ACM, New York, NY, USA, 26 pages. <https://doi.org/10.1145/3519939.3523444>

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

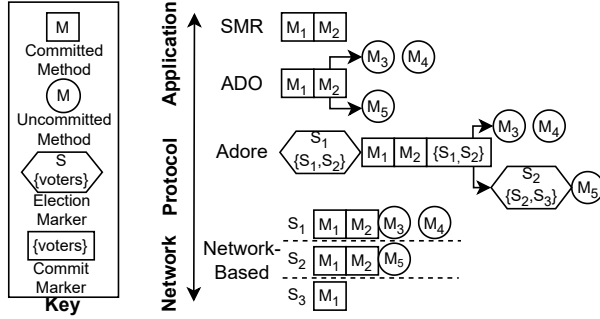


Figure 1. The spectrum of distributed system models. Each shows a snapshot of the same distributed state at a different level of abstraction. Network-based represents each node (S_1, S_2, S_3) as a separate object. SMR and ADO treat the committed methods as a single, atomic object, but ADO also exposes uncommitted methods. ADORE is like ADO, but with additional evidence of the system’s safety in the form of election and commit markers that record the servers that approved each operation.

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 Appendix A 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 remain 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,

```

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

1 // Network
2 for s in cfg { send(s, ELECT); }
3 votes = [];
4 for s in cfg { if recv(s) == VOTE { votes.add(s); } }
5 if !isQuorum(votes) { return FAIL; }
6 for s in cfg { send(s, COMMIT, ["put", "a", 1]); }
7 votes = [];
8 for s in cfg { if recv(s) == COMMITTED { votes.add(s); } }
9 if isQuorum(votes) { return OK; } else { return FAIL; }

1 // ADO
2 if !pull() { return FAIL; }
3 if !invoke(["put", "a", 1]) { return FAIL; }
4 if push() { return OK; } else { return FAIL; }

```

Figure 2. Pseudocode representing the client-facing interfaces for updating a distributed key-value store in three models.

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

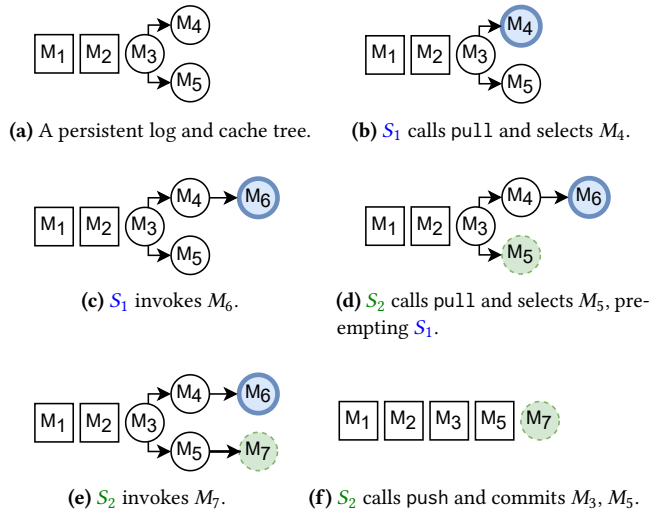


Figure 3. Sample ADO behaviors.

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 $P \bullet C$ 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 M_1 and M_2 , so they are committed. This is reflected in the ADO model’s persistent log by the squares. S_1 and S_2 have M_3 and M_5 in the third slot, respectively. Since one replica is not a quorum in a three-replica configuration, both M_3 and M_5 are uncommitted, hence the circles in the ADO cache tree. As S_1 and S_2 disagree about the methods in this third slot, M_3 and M_5 are represented in different branches in the tree. M_4 is uncommitted and comes after M_3 in S_1 , so it is placed on the same branch as M_3 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 records the caller as the leader at that time. It also selects an

arbitrary cache to serve as the leader’s *active cache*. For example, in Fig. 3b, S_1 ’s active cache is the blue one containing M_4 . A failed pull represents a candidate that did not receive a quorum of votes; however, it may still block leaders with smaller timestamps from committing new methods.

Method Invocation. Method invocation adds a child to the caller’s active cache, and sets the new cache as active (e.g., S_1 and M_6 in Fig. 3c). Only leaders are allowed to invoke methods, but multiple leaders can be active at the same time (e.g., S_1 and S_2 in Figs. 3d and 3e). 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. 3f, S_2 only manages to commit M_3 and M_5 , leaving M_7 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

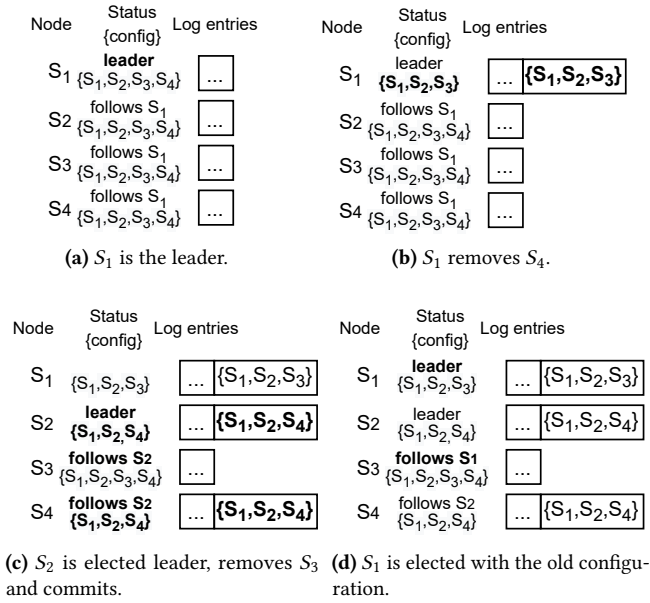


Figure 4. Raft’s reconfiguration can violate safety.

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 $\{S_1, S_2, S_4\}$). Suppose then S_1 initiates another election and receives votes from itself and S_3 . Because it also uses the latest configuration from its log ($\{S_1, S_2, S_3\}$), these two votes constitute a quorum and it wins the election. At this point the game is lost because S_1 and S_2 are leaders with disjoint quorums, which means 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 the effect this has on elections. In particular, although R2 prevents a leader from changing the configuration until the current one is committed, it does not stop other leaders from being elected under uncommitted configurations.

The Solution. The solution proposed by Ongaro [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

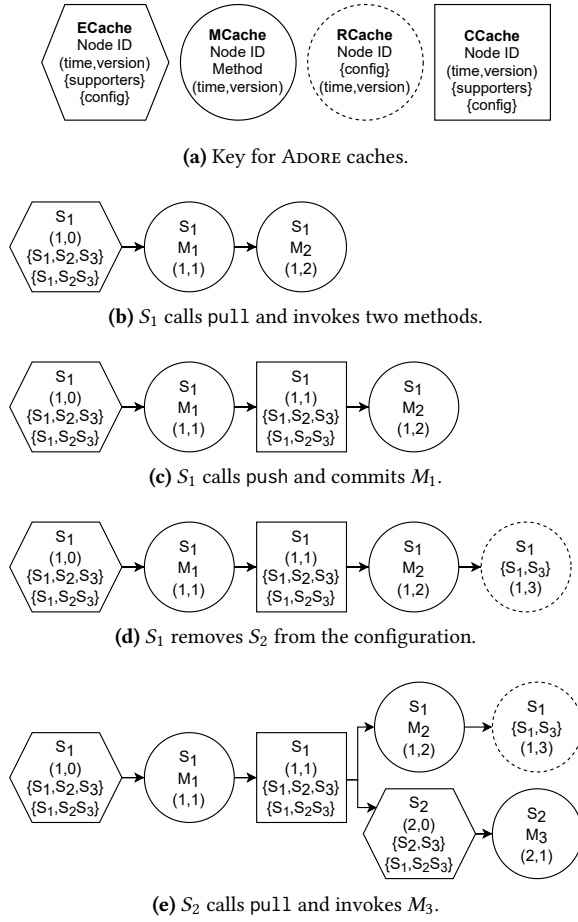


Figure 5. Sample ADORE behaviors.

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, S_2 and S_3 , have observed S_1 ’s *MCache* or *RCache* yet. Note that unlike Fig. 1 where committed methods are represented by squares, *MCaches* and *RCaches* 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

$$\begin{aligned}
 \text{Cache} &\triangleq \text{ECache}(\mathbb{N}_{nid} * \mathbb{N}_{time} * \mathbb{N}_{orsn} * \text{Set}(\mathbb{N}_{nid}) * \boxed{\text{Config}}) \\
 &| \text{MCache}(\mathbb{N}_{nid} * \mathbb{N}_{time} * \mathbb{N}_{orsn} * \text{Method} * \boxed{\text{Config}}) \\
 &| \boxed{\text{RCache}(\mathbb{N}_{nid} * \mathbb{N}_{time} * \mathbb{N}_{orsn} * \text{Config})} \\
 &| \text{CCache}(\mathbb{N}_{nid} * \mathbb{N}_{time} * \mathbb{N}_{orsn} * \text{Set}(\mathbb{N}_{nid}) * \boxed{\text{Config}}) \\
 \text{CacheTree} &\triangleq \mathbb{N}_{cid} \rightarrow \mathbb{N}_{cid} * \text{Cache} \\
 \text{TimeMap} &\triangleq \mathbb{N}_{nid} \rightarrow \mathbb{N}_{time} \\
 \Sigma_{\text{Adore}} &\triangleq \text{CacheTree} * \text{TimeMap}
 \end{aligned}$$

Figure 6. ADORE state definitions.

as structural invariants. This also makes it easy to see the mutual dependency between *RCaches*, *ECaches*, and *CCaches* 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 pull, push, invoke, and reconfig operations map fairly directly onto the election, commit, and local log update phases found in most consensus protocols, so in fact, this relation can be proved for many protocols, including various Paxos variants and Raft. Section 5 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 (see Section 6 for examples).

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 Σ_{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

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

Assumptions about $R1^+$ and $isQuorum$

(REFLEXIVE) $R1^+(cf, cf)$

(OVERLAP) $R1^+(cf, cf') \wedge isQuorum(Q, cf) \wedge isQuorum(Q', cf')$
 $\implies Q \cap Q' \neq \emptyset$

Definitions

$R2(tr, C) \triangleq \forall C' \in tr. C' = RCache(_) \wedge C' \uparrow C \implies$

$\exists C'' \in tr. C'' = CCache(_) \wedge C' \uparrow C'' \wedge C'' \uparrow C$

$R3(tr, C) \triangleq \exists C' \in tr.$

$C' = CCache(_) \wedge time(C') = time(C) \wedge C' \uparrow C$

$canReconf(tr, C, ncf) \triangleq R1^+(conf(C), ncf) \wedge R2(tr, C) \wedge R3(tr, C)$

Figure 7. Configuration/quorum parameters and definitions.

$name(st)$ to represent extracting one of these fields (e.g., $tree(st)$ returns the first element). Fig. 7 declares that the type of the configuration ($Config$) is an opaque parameter with functions to extract a set of nodes ($mbrs$) and decide if some set constitutes a quorum ($isQuorum$). This allows the model and safety proof to work for any instantiation of these parameters as long as they satisfy the REFLEXIVE and OVERLAP invariants ($C' \uparrow C$ means C' is an ancestor of C).

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 $CCaches$ lie on the same branch of the cache tree, which implies global agreement on a consistent commit history.

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

Each type of cache contains the node ID (nid) of the replica that called the operation creating it ($caller$), a timestamp ($time$), a version number ($versn$), and the configuration ($conf$) under which it was called (the root cache is initialized with some $conf_0$). The timestamp corresponds to a Paxos ballot number or a Raft term number and is assigned to a leader in each round. The version number resets to 0 at the start of each round and increments on every method/reconfig call.

$Op \triangleq pull : \mathbb{N}_{nid} \rightarrow \Sigma_{Adore} \rightarrow \Sigma_{Adore}$
 $| invoke : \mathbb{N}_{nid} \rightarrow Method \rightarrow \Sigma_{Adore} \rightarrow \Sigma_{Adore}$
 $| reconfig : \mathbb{N}_{nid} \rightarrow Config \rightarrow \Sigma_{Adore} \rightarrow \Sigma_{Adore}$
 $| push : \mathbb{N}_{nid} \rightarrow \Sigma_{Adore} \rightarrow \Sigma_{Adore}$

Figure 8. ADORE operations.

$C_1 > C_2 \triangleq (time(C_1), versn(C_1)) > (time(C_2), versn(C_2))$

$\vee ((time(C_1), versn(C_1)) = (time(C_2), versn(C_2)))$

$\wedge C_1 = CCache(_) \wedge C_2 \neq CCache(_)$

$setTimes(st, Q, t) \triangleq (tree(st), times(st)[s \mapsto t \mid \forall s \in Q])$

$isLeader(st, nid, t) \triangleq times(st)[nid] = t$

$validSupp(nid, Q, C) \triangleq nid \in Q \wedge Q \subseteq mbrs(conf(C))$

$mostRecent(tr, Q) \triangleq \max_{>} \{C \in tr \mid Q \cap supporters(C) \neq \emptyset\}$

$activeCache(tr, nid) \triangleq \max_{>} \{C \in tr \mid caller(C) = nid\}$

$lastCommit(tr, nid) \triangleq \max_{>} \left\{ C \in tr \left| \begin{array}{l} nid \in supporters(C) \\ \wedge C = CCache(_) \end{array} \right. \right\}$

$canCommit(C, nid, st) \triangleq$

$(C = MCache(_) \vee C = RCache(_)) \wedge caller(C) = nid$

$\wedge isLeader(st, nid, time(C)) \wedge C > lastCommit(tree(st), nid)$

Figure 9. Selected ADORE auxiliary definitions.

In Fig. 9, we define a strict order ($>$) on caches by comparing the lexicographic order of their timestamp and version number pairs, with the exception that if a $CCache$ has the same timestamp and version as a non- $CCache$, the $CCache$ is considered greater, which is needed to make $>$ a total order.

$ECaches$ and $CCaches$ also carry the node IDs of their $supporters$; i.e., the replicas that voted for them. An $MCache$ or $RCache$'s only $supporter$ is its caller. $MCache$ s also contain the name of a $Method$. 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.

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

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.

$$\begin{array}{c}
\text{PULLOK} \\
\frac{\textcircled{O} \text{pull}(st, nid) = Ok(Q, Q_{ok}, C_{max}, t) \quad st' \triangleq setTimes(st, Q, t) \\
C_{new} \triangleq ECache(nid, t, 0, Q, conf(C_{max}))}{\textcircled{O} \vdash \text{pull}(nid) : st \rightsquigarrow \text{if } Q_{ok} \text{ then } addLeaf(st', C_{max}, C_{new}) \text{ else } st'} \\
\\
\text{INVOKEOK} \\
\frac{C_A \triangleq activeCache(tree(st), nid) \quad isLeader(st, nid, time(C_A)) \\
C_{new} \triangleq MCache(nid, time(C_A), vrsn(C_A) + 1, M, conf(C_A))}{\textcircled{O} \vdash \text{invoke}(nid, M) : st \rightsquigarrow addLeaf(st, C_A, C_{new})} \\
\\
\text{RECONFIGOK} \\
\frac{C_A \triangleq activeCache(tree(st), nid) \quad isLeader(st, nid, time(C_A)) \\
canReconf(tree(st), C_A, ncf) \\
C_{new} \triangleq RCache(nid, time(C_A), vrsn(C_A) + 1, ncf)}{\textcircled{O} \vdash \text{reconfig}(nid, ncf) : st \rightsquigarrow addLeaf(st, C_A, C_{new})} \\
\\
\text{PUSHOK} \\
\frac{\textcircled{O} \text{push}(st, nid) = Ok(Q, Q_{ok}, C_M) \quad st' \triangleq setTimes(st, Q, time(C_M)) \\
C_{new} \triangleq CCache(nid, time(C_M), vrsn(C_M), Q, conf(C_M))}{\textcircled{O} \vdash \text{push}(nid) : st \rightsquigarrow \text{if } Q_{ok} \text{ then } insertBtw(st', C_M, C_{new}) \text{ else } st'} \\
\\
\text{PULLNOOP} \\
\frac{\textcircled{O} \text{pull}(st, nid) = Fail}{\textcircled{O} \vdash \text{pull}(nid) : st \rightsquigarrow st} \quad \text{Similar NOOP rules exist for the other operations. See the Appendix.}
\end{array}$$

Figure 10. Semantics of ADORE operations.

$$\begin{array}{c}
\text{VALIDPULLORACLE} \\
\frac{validSupp(nid, Q, C_{max}) \quad Q_{ok} \triangleq isQuorum(Q, conf(C_{max})) \\
\forall s \in Q. times(st)[s] < t \quad C_{max} \triangleq mostRecent(tree(st), Q)}{\textcircled{O} \text{pull}(st, nid) = Ok(Q, Q_{ok}, C_{max}, t)} \\
\\
\text{VALIDPUSHORACLE} \\
\frac{validSupp(nid, Q, C_M) \quad Q_{ok} \triangleq isQuorum(Q, conf(C_M)) \\
\forall s \in Q. times(st)[s] \leq time(C_M) \quad canCommit(C_M, nid, st)}{\textcircled{O} \text{push}(st, nid) = Ok(Q, Q_{ok}, C_M)}
\end{array}$$

Figure 11. Valid pull and push oracle conditions.

ADORE models this with `pull`, which relies on $\textcircled{O} \text{pull}$ to simulate the network and arbitrarily decide what set of supporters (Q) receive the election request. One can imagine the oracle as an omnipotent observer that abstracts over the network details by treating it as a nondeterministic black box. Fig. 11 defines conditions that a valid oracle must satisfy.

On success the oracle chooses a set of supporters and a time (t) that is strictly larger than any they have previously observed. The cache it returns (C_{max}) is the result of *mostRecent*, and is the most up-to-date cache supported by any replica in Q . This guarantees that the leader learns about every committed method. The supporters' timestamps are updated to reflect their vote. There are two outcomes for the cache tree depending on Q . If it is not a quorum then the election fails and the only effect is the change in the timestamps.

Otherwise, a new *ECache* child is added to C_{max} . The oracle may also return failure in which case the state is unchanged (see `PULLNOOP` 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 (C_A), 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 essentially a special kind of *MCache* that contains a new configuration (ncf) instead of a method. *RCaches* are created by `reconfig`, whose semantics are nearly identical to regular method invocation, except for a few additional restrictions, which are modified versions of R1–3 from Section 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 ADO model, a successful push commits an arbitrary prefix of the most recent uncommitted commands. This is chosen by $\textcircled{O} \text{push}$, which returns a cache (C_M) that satisfies *canCommit*. This means C_M must be an *MCache* or *RCache* that was called by nid with its current timestamp, and is more recent than the latest *CCache* supported by nid . This guarantees that nid is a valid leader, C_M is an uncommitted command, and committing it will not conflict with a previous commit. Like `pull`, the supporters' timestamps must not be greater than C_M 's, though they may be equal.

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

4 Safety Proof

The primary purpose of ADORE is to simplify the verification of safety properties of consensus protocols even with the complexity introduced by reconfiguration. This section demonstrates how it accomplishes this goal by sketching the proof of replicated state safety and highlighting interesting challenges. For reasons of clarity and space, we stick

mainly to informal arguments, but more rigorous proofs can be found in both Appendix B 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.¹ 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_{C_1} and C_{C_2} , 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.

¹Certain variants of multi-Paxos allow committing slots while earlier entries are still undecided, but this property still holds once the gaps are filled in.

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.

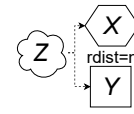
The typical approach to proving safety in a network-based model goes by induction over the trace of network events; i.e., assume the property holds, then show it continues to hold when some replica receives a commit request, or when a candidate receives a quorum of election votes, and so on.

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 Appendix B. 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



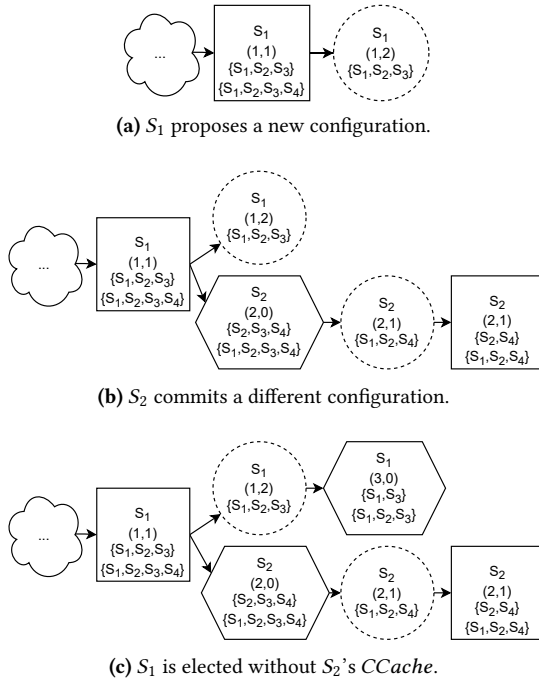


Figure 12. An example of a breach of safety without R3.

in that it prevents a leader from beginning a new reconfiguration while another leader still has one in progress. This is a particularly subtle problem because the new leader might not even be aware of the old reconfiguration.

For example, consider the situation in Fig. 12 (this is the same as Fig. 4 but with cache trees). The leader S_1 removes S_4 from the configuration but fails to commit it. Then S_2 becomes the leader, but is unaware of the reconfiguration attempt because its supporters do not include S_1 , so it begins its own reconfiguration by removing S_3 . It succeeds with a quorum (S_2 and S_4) of supporters. At this point any future election must have this *CCache* in its history or else safety is compromised. However, because S_1 and S_3 did not participate in S_2 's reconfiguration, S_1 is elected using its own *RCache* because S_2 's *CCache* has a larger timestamp.

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

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

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

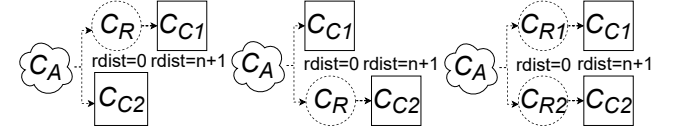
4.3 General Case

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

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

Theorem 4.5 (Safety). *Any cache tree tr with any $rdist$ satisfies replicated state safety.*

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



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

5 Refinement

We now know that 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 ADORE and these protocols and show that ADORE's safety guarantees their safety. We demonstrate this for a slightly simplified version of Raft, but note that this is just one of many possible implementations.

In a sense, ADORE models an abstract, synchronous version of Raft in which communication happens atomically and in logical time order. Through a series of refinements we show that a more realistic asynchronous network-based

$$\begin{aligned} \Sigma_{\text{net}} &\triangleq (\mathbb{N}_{\text{nid}} \rightarrow \text{Server}) * \text{Network} \\ \text{Server} &\triangleq \mathbb{N}_{\text{time}} * \mathbb{N}_{\text{orsn}} * \text{List}(\mathbb{N}_{\text{time}} * \text{Method} * \text{Config}) * \dots \\ \text{Network} &\triangleq \text{Set}(\text{Msg}) * \text{Set}(\text{Msg}) \\ \text{Op}_{\text{net}} &\triangleq \text{elect} : \mathbb{N}_{\text{nid}} \rightarrow \Sigma_{\text{net}} \rightarrow \Sigma_{\text{net}} \\ &\quad | \text{commit} : \mathbb{N}_{\text{nid}} \rightarrow \Sigma_{\text{net}} \rightarrow \Sigma_{\text{net}} \\ &\quad | \text{invoke} : \mathbb{N}_{\text{nid}} \rightarrow \text{Method} \rightarrow \Sigma_{\text{net}} \rightarrow \Sigma_{\text{net}} \\ &\quad | \text{reconfig} : \mathbb{N}_{\text{nid}} \rightarrow \text{Config} \rightarrow \Sigma_{\text{net}} \rightarrow \Sigma_{\text{net}} \\ &\quad | \text{deliver} : \text{Msg} \rightarrow \Sigma_{\text{net}} \rightarrow \Sigma_{\text{net}} \end{aligned}$$

Figure 13. Selected Raft network-based state and operations.

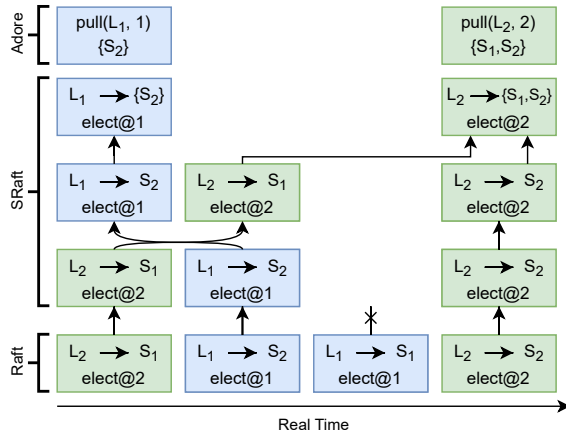


Figure 14. Raft to SRAFT to ADORE refinement. $L_1 \rightarrow S_1$ denotes a message sent from a leader to a server with the type and logical timestamp of the message indicated below.

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 `reconfig`, which are local operations that only affect its own log.

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

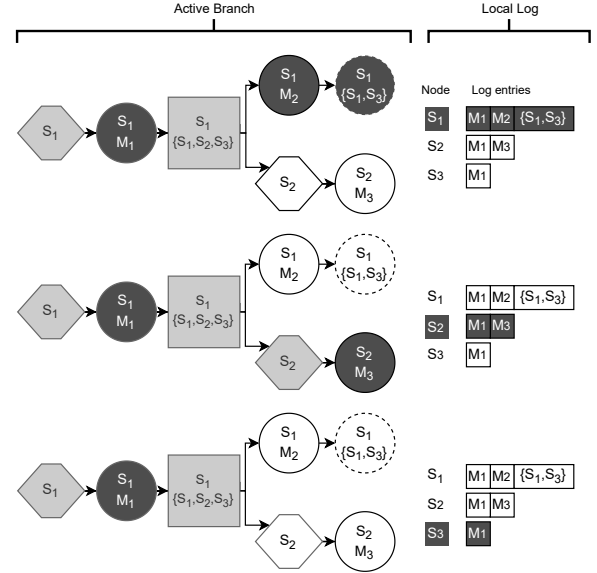


Figure 15. Correspondence between replicas' local logs and active branches. The active branch of each replica is shown by the gray backgrounds, with the dark gray indicating caches that have matching entries in the local log (MCaches and RCaches).

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 \mathbb{R} between Raft and ADORA's state. This includes correspondences between auxiliary state such as timestamps, and leadership flags, but for ADORA's safety to imply something useful about Raft, \mathbb{R} must ensure that for any replica's local log, the methods are the same as those along that replica's active branch of the cache tree. Then, since ADORA guarantees that every replica's active branch has a common prefix of committed methods, the same must be true in Raft as long as \mathbb{R} holds. SRAFT and Raft share the same state definitions, so \mathbb{R} also holds for SRAFT and ADORA.

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

Intuitively, the correspondence between ADORA and Raft steps is clear: `pull` for elections, `push` for commits, and ADORA method invocation and reconfiguration for their Raft counterparts. The reality is less straightforward because the ADORA 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 \mathbb{R} . Because we have already “lined up” the events, the bulk of the remaining work is to translate between different state representations, such as a replica's local log in SRAFT and its active branch in ADORE (Fig. 15). See Appendix C 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 $R1^+$ have a non-empty intersection. This means that for any valid instantiation of the configuration-related parameters the safety proof holds for free. To give a sense of how flexible ADORE's reconfiguration scheme is, we demonstrate several practical and diverse implementations.

Raft Single-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.

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

Raft Joint Consensus. A more complicated case is Raft's other reconfiguration strategy [26], which allows arbitrary configuration changes. Like the single-node version, a new configuration is proposed as a special command, but instead of immediately switching to the new configuration, the replicas transition to an intermediate “joint configuration” consisting of both the old and new members. In this state elections and commit operations require support from majorities of *both* configurations (not their union) to succeed. Once a command is committed under the joint configuration it is safe to transition to the new configuration.

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

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

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

In this case the primary is constant and is a member of every quorum, so all quorums obviously intersect. The limitation of this is if the primary crashes then all progress is blocked. A more reliable alternative is to use one of the previous approaches to manage a set of primaries that can be replaced as needed. 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.

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

The argument for why this guarantees overlap is a generalization of the single-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

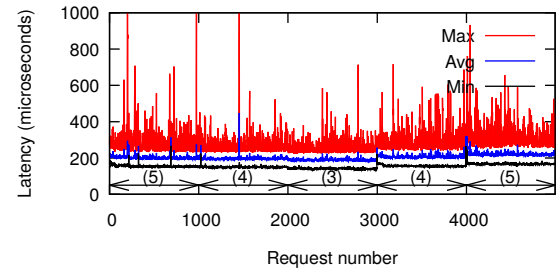


Figure 16. OCaml Raft performance under reconfiguration. (n) indicates a configuration under n nodes.

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 α such that a configuration committed in instance i takes effect in instance $i + \alpha$, thus allowing i through $i + \alpha - 1$ to continue as normal.

This 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 α 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]. IronFleet’s Paxos-based IronRSL 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

- [1] 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>
- [2] 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>
- [3] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernan Vanzetto. 2012. TLA+ Proofs. In *Proc. of the 18th International Symposium on Formal Methods (FM '12, Vol. 7436)*, Dimitra Giannakopoulou and Dominique Mery (Eds.). Springer-Verlag, Berlin, Heidelberg, 147–154. <https://www.microsoft.com/en-us/research/publication/tla-proofs/>
- [4] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. 2022. Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback. In *Proc. of the 26th International Conference on Financial Cryptography and Data Security* (Grenada) (FC '22). Springer-Verlag, Berlin, Heidelberg. <https://fc22.ifca.ai/preproceedings/35.pdf>
- [5] 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>
- [6] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffrey 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>
- [7] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proc. of the 25th Symposium on Operating Systems Principles* (Monterey, CA, USA) (SOSP '15). ACM, New York, NY, USA, 1–17. <https://doi.org/10.1145/2815400.2815428>
- [8] Wolf Honoré, Jieung Kim, Ji-Yong Shin, and Zhong Shao. 2021. Much ADO about Failures: A Fault-Aware Model for Compositional Verification of Strongly Consistent Distributed Systems. *Proc. ACM Program. Lang.* 5, OOPSLA (Oct. 2021). <https://doi.org/10.1145/3485474>
- [9] Wolf Honoré, Ji-Yong Shin, Jieung Kim, and Zhong Shao. 2022. *Adore: Atomic Distributed Objects with Certified Reconfiguration*. Technical Report YALEU/DCS/TR-1560. Yale Univ. <https://flint.cs.yale.edu/publications/adore.html>
- [10] Wolf Honoré, Ji-Yong Shin, Jieung Kim, and Zhong Shao. 2022. *Artifact For "Adore: Atomic Distributed Objects with Certified Reconfiguration"*. Yale University. <https://doi.org/10.5281/zenodo.6321150>
- [11] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proc. of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA, USA) (USENIXATC '10). USENIX Association, Berkeley, CA, USA, 11. <https://doi.org/10.5555/1855840.1855851>
- [12] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169. <https://doi.org/10.1145/279227.279229>
- [13] Leslie Lamport. 1999. Specifying Concurrent Systems with TLA+. *Calcutational System Design* (April 1999), 183–247. <https://www.microsoft.com/en-us/research/publication/specifying-concurrent-systems-tla/>
- [14] Leslie Lamport. 2001. Paxos Made Simple. *SIGACT News* 32, 4 (Dec. 2001), 51–58. <https://doi.org/10.1145/568425.568433>
- [15] Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston, MA, USA. <https://www.microsoft.com/en-us/research/publication/specifying-systems-the-tla-language-and-tools-for-hardware-and-software-engineers/>
- [16] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2008. *Reconfiguring a State Machine*. Technical Report MSR-TR-2008-193. Microsoft. <https://www.microsoft.com/en-us/research/publication/reconfiguring-a-state-machine/>
- [17] 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>
- [18] Barbara Liskov and James Cowling. 2012. *Viewstamped Replication Revisited*. Technical Report MIT-CSAIL-TR-2012-021. MIT. <http://hdl.handle.net/1721.1/71763>
- [19] Giuliano Losa and Mike Dodds. 2020. On the Formal Verification of the Stellar Consensus Protocol. In *Proc. of the 2nd Workshop on Formal Methods for Blockchains (FMBC '20, Vol. 84)*, Bruno Bernardo and Diego Marmosler (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1–9. <https://doi.org/10.4230/OASlcs.FMBC.2020.9>
- [20] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasicki, 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>
- [21] Dahlia Malkhi, Leslie Lamport, and Lidong Zhou. 2008. *Stoppable Paxos*. Technical Report MSR-TR-2008-192. Microsoft. <https://www.microsoft.com/en-us/research/publication/stoppable-paxos/>
- [22] David Mazieres. 2015. The Stellar Consensus Protocol: A Federated Model for Internet-Level Consensus. <https://www.stellar.org/papers/stellar-consensus-protocol>
- [23] 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>
- [24] Diego Ongaro. 2014. *Consensus: Bridging Theory and Practice*. Ph.D. Dissertation. Stanford University.
- [25] Diego Ongaro. 2015. bug in single-server membership changes. <https://groups.google.com/g/raft-dev/c/t4xj6dJTP6E/m/d2D9LrWRza8J>
- [26] 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>
- [27] 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>
- [28] 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>
- [29] 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>
- [30] 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>

- [31] 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>
- [32] William Schultz, Ian Dardik, and Stavros Tripakis. 2022. Formal Verification of a Distributed Dynamic Reconfiguration Protocol. In *Proc. of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Philadelphia, PA, USA) (*CPP '22*). ACM, New York, NY, USA, 143–152. <https://doi.org/10.1145/3497775.3503688>
- [33] William Schultz, Siyuan Zhou, Ian Dardik, and Stavros Tripakis. 2022. Design and Analysis of a Logless Dynamic Reconfiguration Protocol. In *Proc. of the 25th International Conference on Principles of Distributed Systems* (Strasbourg, France) (*OPODIS '21, Vol. 217*), Quentin Bramas, Vincent Gramoli, and Alessia Milani (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1–16. <https://doi.org/10.4230/LIPIcs.OPODIS.2021.26>
- [34] 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>
- [35] 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>
- [36] The Coq Development Team. 1999–2022. The Coq Proof Assistant. <http://coq.inria.fr>.
- [37] 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>
- [38] 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>
- [39] 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>
- [40] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. Planning for Change in a Formal Verification of the Raft Consensus Protocol. In *Proc. of the 5th ACM SIGPLAN International Conference on Certified Programs and Proofs* (St. Petersburg, FL, USA) (*CPP '16*). ACM, New York, NY, USA, 154–165. <https://doi.org/10.1145/2854065.2854081>
- [41] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proc. of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) (*PODC '19*). ACM, New York, NY, USA, 347–356. <https://doi.org/10.1145/3293611.3331591>

A Consensus Primer


Both the election and commit phases involve a replica broadcasting a message to the others and trying to collect positive acknowledgements from a quorum. The goal of the election phase is for a replica, known as a *candidate*, to get permission to add new log entries by confirming that its has all of the existing committed entries. This requires a way to determine which of two logs is more “up to date”. Replicas do not have access to a synchronized global clock, so instead, every command is assigned a logical *timestamp* (called a ballot number in Paxos and a term number in Raft) and logs are compared by looking at the timestamps of their last entries.

Replicas keep track of the largest timestamp they have observed and only vote for candidates with larger timestamps and sufficiently up-to-date logs. A candidate wins an election and becomes a leader if it receives a quorum of votes, at which point it moves on to the commit phase.


Paxos and Raft use different approaches to ensure that a candidate’s log is sufficiently up-to-date, but the result is the same in both cases. In Paxos, replicas respond to the candidate with their own logs, and the candidate chooses the one whose last entry has the latest timestamp. A candidate in Raft sends its log to the replicas, which compare against their own logs to decide how to vote. If a candidate receives a quorum of positive votes it becomes a leader and moves on to the commit phase. Although the place at which the comparison is done differs, at the end of a successful election the leader’s log is guaranteed to contain the most up-to-date log out of a quorum of replicas (at least at the time that the replicas voted).

During the commit phase, the leader handles client requests by appending commands to its log and asking the replicas to do the same. If a replica still believes that the leader’s log is sufficiently up-to-date and has not been preempted by a newer leader it adds the command to its log. When a quorum of replicas accept the command it is committed and the leader can inform the client. A leader continues in the commit phase until it learns that there is a newer leader, at which point it steps down and behaves as a regular replica. If a replica suspects the leader has crashed (e.g., because it has been too long since it last received a message), it may begin a new election.

B Additional Safety Proofs


This section provides the omitted proofs from Section 4. For each theorem its corresponding  [Coq theorem name] is included after the theorem’s statement. The Coq theorems and proofs can be found in the source code [10].

Lemma B.1 (Descendant Order). *If C_Y is a descendant of C_X then $C_Y > C_X$.*

 [rado_inv_descendant_lt]


PROOF. It is enough to show that every newly added cache is greater than its parent. An *ECache* added by pull has a larger timestamp than its supporters, including its parent. *MCaches* and *RCaches* increment their parent’s version number. push copies the parent’s time and version, but since the parent must be an *MCache* or *RCache*, the *CCache* is greater by definition of $>$. \square

Lemma B.2 (Leader Time Uniqueness, $\text{rdist}=0$). *If C_{E1} and C_{E2} are *ECaches* and $\text{rdist}(C_{E1}, C_{E2}) = 0$, then $\text{time}(C_{E1}) \neq \text{time}(C_{E2})$.*

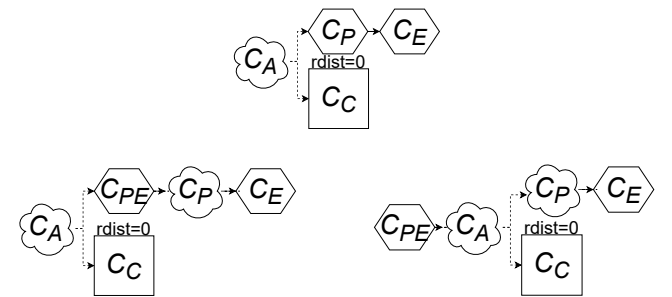
 [rado_inv_E_unique_time_no_R]

PROOF. Because $\text{rdist} = 0$, C_{E1} and C_{E2} have the same configuration and thus overlapping quorums of supporters. pull chooses a time greater than any observed by its supporters, so, since C_{E1} and C_{E2} share a supporter, whichever cache was added to the tree last must have a larger timestamp. \square

Theorem B.3 (Election-Commit Order, $\text{rdist}=0$). *Let C_C be a *CCache* and C_E be an *ECache* such that $C_E > C_C$ and $\text{rdist}(C_E, C_C) = 0$. C_E must be a descendant of C_C .*

 [rado_inv_EC_descendant_no_R]

PROOF. If C_E is a descendant of C_C then we are done, so suppose it is not for the sake of deriving a contradiction. By Lemma B.1, C_C cannot be a descendant of C_E either. By double induction on the time and version number, we can assume that C_E is the first *ECache* that is not a descendant of C_C and for all *ECaches* C'_E where $C_E > C'_E > C_C$, C'_E is a descendant of C_C . Because $\text{rdist} = 0$, C_E and C_C have the same configuration and thus overlapping quorums of supporters. Therefore C_E ’s parent, C_P , is greater than C_C because pull selects the largest cache supported by its supporters. This leaves the three following options for the shape of the tree:



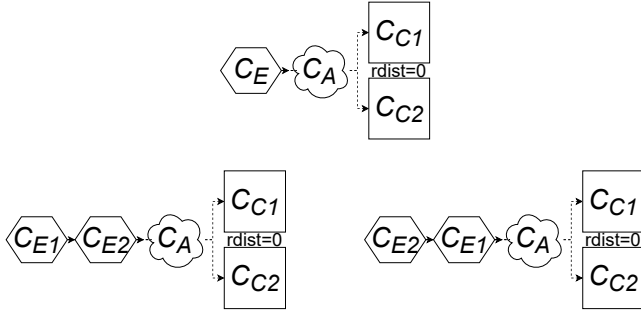
In the first case C_P is an *ECache*, but then $C_E > C_P > C_C$, so by the inductive hypothesis it must be a descendant of C_C , which is a contradiction. If C_P is not an *ECache*, it must have an *ECache* ancestor, C_{PE} , such that $\text{time}(C_P) = \text{time}(C_{PE})$. From $C_P > C_C$ we know $\text{time}(C_P) \geq \text{time}(C_C)$, but thanks to Lemma B.2, we also know that if $\text{time}(C_P) = \text{time}(C_C)$ they must be on the same branch. Since they are not, $\text{time}(C_P) = \text{time}(C_{PE}) > \text{time}(C_C)$ so $C_{PE} > C_C$.

The two possible locations for C_{PE} are between the common ancestor C_A and C_P , or before C_A . The first option derives a contradiction because $C_E > C_{PE} > C_C$, but C_{PE} is not a descendant of C_C . The second case is also impossible by Lemma B.1 because C_{PE} is an ancestor of C_C , but $C_{PE} > C_C$. Thus, it is impossible to arrive at a cache tree in which C_E is not a descendant of C_C . \square

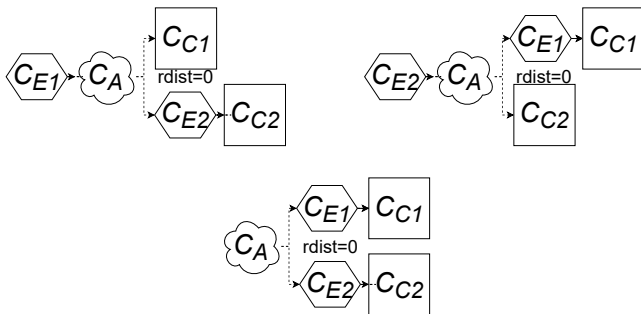
Theorem B.4 (Safety, $rdist=0$). *Let C_{C1} and C_{C2} be CCaches such that $rdist(C_{C1}, C_{C2}) = 0$. Either C_{C1} is a descendant of C_{C2} or C_{C2} is a descendant of C_{C1} .*

$\text{[rado_inv_C_linear_no_R]}$

PROOF. As before, assume that neither C_{C1} nor C_{C2} is a descendant of the other in order to derive a contradiction. Each must have a nearest *ECache* ancestor, C_{E1} and C_{E2} respectively, with the same time and no intervening *ECaches*. Three possibilities are as follows:



In the first case C_{E1} and C_{E2} are the same *ECache*, called C_E , and in the other two the elections are distinct common ancestors of C_{C1} and C_{C2} . Recall that there can be no *ECaches* between C_{E1} and C_{C1} and likewise for C_{E2} and C_{C2} so the latter two cases are impossible. Similarly, in the first case there are no *ECaches* anywhere on the forking branches after C_A , but this is also impossible because pull is the only operation that can create forks in the tree. This leaves three options:



In the first case C_{E2} is a descendant of C_{E1} so $C_{E2} > C_{E1}$ by Lemma B.1. This also implies $time(C_{E2}) > time(C_{E1})$

because an *ECache*'s version number is always 0. Then, because $time(C_{E1}) = time(C_{C1})$, $C_{E2} > C_{C1}$ as well. By Theorem B.3, C_{E2} must be a descendant of C_{C1} , but it is not, so this case is impossible. The second case follows by a symmetric argument. The final case also contradicts Theorem B.3 if $time(C_{E1}) \neq time(C_{E2})$, which we know is true by Lemma B.2. \square

Lemma B.5 (Leader Time Uniqueness, $rdist=1$). *If C_{E1} and C_{E2} are ECaches and $rdist(C_{E1}, C_{E2}) = 1$, then $time(C_{E1}) \neq time(C_{E2})$.*

$\text{[rado_inv_E_unique_time_overlap]}$

PROOF. $R1^+$ guarantees that quorums of C_{E1} and C_{E2} 's configurations overlap. pull chooses a time greater than any observed by its supporters, so, since C_{E1} and C_{E2} share a supporter, whichever cache was added to the tree last must have a larger timestamp. \square

Theorem B.6 (Election-Commit Order, $rdist=1$). *Let C_C be a CCache and C_E be an ECACHE such that $C_E > C_C$ and $rdist(C_E, C_C) = 1$. C_E must be a descendant of C_C .*

$\text{[rado_inv_ERC_descendant_no_R]}$

$\text{[rado_inv_REC_descendant_no_R]}$

PROOF. The proof of Theorem B.3 relied on Lemma B.1 and Lemma B.2 to show that every case where C_E is not a descendant of C_C is impossible. The only difference in this case is that $rdist = 1$; however, Lemma B.1 is independent of $rdist$, and Lemma B.2 can be replaced by Lemma B.5, so nearly exactly the same proof as before works. \square

Theorem B.7 (Safety, $rdist=1$). *Let C_{C1} and C_{C2} be CCaches such that $rdist(C_{C1}, C_{C2}) = 1$. Either C_{C1} is a descendant of C_{C2} or C_{C2} is a descendant of C_{C1} .*

$\text{[rado_inv_RC_linear_no_ERC]}$

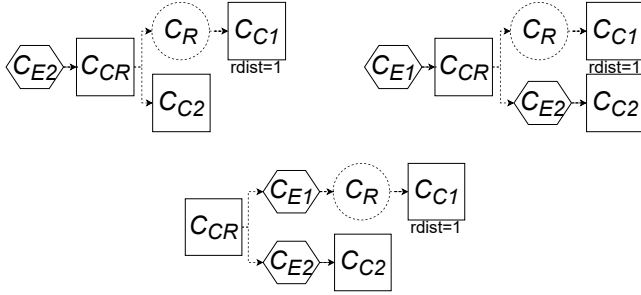
$\text{[rado_inv_RC_linear_no_RC]}$

PROOF. If C_{C1} and C_{C2} are on the same branch then we are done, so suppose they are not for the sake of deriving a contradiction and let C_A be a common ancestor. Because $rdist = 1$ there is one *RCache*, C_R , between either C_A and C_{C1} or C_A and C_{C2} . Without loss of generality, suppose it is on C_{C1} 's branch. By R3 C_R must have a *CCache* ancestor, C_{CR} with the same time and no other intervening *CCaches*. The possible locations for C_{CR} are:



The first case has two *CCaches* with $rdist = 0$ on separate branches, which is impossible by Theorem B.4. In the second case C_{CR} is a common ancestor of C_R and C_{C2} . Each

$CCache$ must have a nearest $ECache$ ancestor, C_{E1} and C_{E2} respectively, with the same time and no intervening $ECaches$. Three possibilities are as follows:

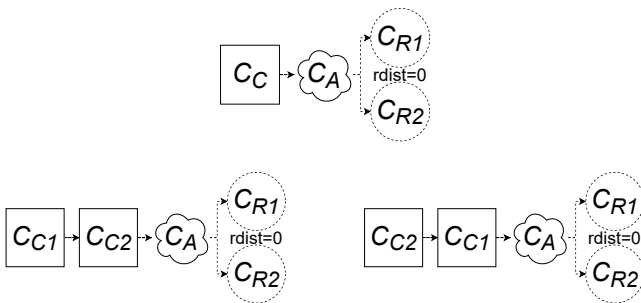


In the first case, recall that $time(C_R) = time(C_{CR})$, which implies that there are no $ECaches$ between them. Likewise, there are none between C_{E2} and C_{C2} , so neither fork has an $ECache$, which is impossible. In the second case, $C_{E2} > C_{E1}$ by Lemma B.1 so $time(C_{E2}) > time(C_{E1})$. Then since $time(C_{E1}) = time(C_R) = time(C_{CR})$, $C_{E2} > C_{CR}$ as well. But this contradicts Theorem B.6 since C_{E2} is not a descendant of C_{CR} . The final case also contradicts Theorem B.6 because $time(C_{E1}) \neq time(C_{E2})$ by Lemma B.5, so one must be greater than the other. This leaves no option but that C_{C1} is a descendant of C_{C2} or vice-versa. \square

Lemma B.8 (CCache in RCache Fork). *Let C_{R1} and C_{R2} be $RCaches$ such that $rdist(C_{R1}, C_{R2}) = 0$, and neither is a descendant of the other, but both have a common ancestor C_A . Then there exists a $CCache$ C_C that is a descendant of C_A and an ancestor of either C_{R1} or C_{R2} .*

\clubsuit [rado_inv_R_branch_case]

PROOF. By R3 each $RCache$ must have a $CCache$ ancestor with the same time, called C_{C1} and C_{C2} respectively. If either C_{C1} or C_{C2} is a descendant of C_A then we are done. Otherwise the options are for C_{C1} and C_{C2} to be the same cache, or for one to be a descendant of the other.



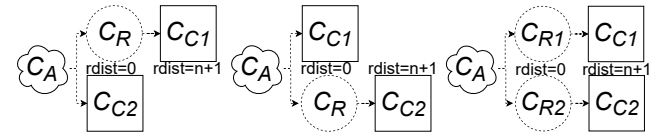
Recall that there cannot be any $ECaches$ between C_{C1} and C_{R1} or between C_{C2} and C_{R2} . This means that in every case that are no $ECaches$ on either forking branch, which is impossible. \square

Theorem B.9 (Safety). *Let tr be a cache tree with any $rdist$. For any $CCaches$ C_{C1} and C_{C2} in tr , one must be a descendant of the other.*

\clubsuit [rado_inv_C_linear]

PROOF. We proceed by induction on $rdist(tr)$. For $rdist \leq 1$ we are done by Theorems B.4 and B.7. Suppose now that all trees with $rdist = n$ are safe, and $1 < rdist(tr) = n + 1$ so $1 < rdist(C_{C1}, C_{C2}) \leq n + 1$. If $rdist(C_{C1}, C_{C2}) \leq n$ then they are in some subtree tr' with $rdist(tr') = n$, so we are done by the inductive hypothesis. Safety also holds if C_{C1} and C_{C2} are on the same branch, and if not we will show that all other shapes for tr are impossible.

There are two options for how the n $RCaches$ could be distributed. Either all $n + 1$ $RCaches$ are on one branch (e.g., $rdist(C_A, C_{C1}) = n + 1$ and $rdist(C_A, C_{C2}) = 0$) or the $RCaches$ are distributed such that there is at least one on both branches. In either case we can identify the first $RCache$ descendant of C_A on both branches.



The first two cases are symmetric, so assume without loss of generality that C_R is on C_{C1} 's branch. Let C_{CR} be the first $CCache$ descendant of C_R . It is enough to show that $rdist(C_{CR}, C_{C2}) \leq n$ because then C_{CR} and C_{C2} must be on the same branch, which is a contradiction. We know $rdist(C_{C1}, C_{C2}) > 1$, so C_R cannot be the only $RCache$ on C_{C1} 's branch. We also know by R2 that this other $RCache$ cannot be between C_R and C_{CR} . Therefore this $RCache$ does not count towards $rdist(C_{CR}, C_{C2})$ and it is at most n .

For the final case, by Lemma B.8 there must be a $CCache$ between C_A and either C_{R1} or C_{R2} . Suppose it is between C_A and C_{R2} and call it C'_{C2} . Now $rdist(C_{R1}, C'_{C2}) = 0$ so this is the same case as before. Therefore this situation is impossible as well and the only possibility is that C_{C1} and C_{C2} are on the same branch. \square

C Refinement

This section contains proof sketches of the key steps of the refinement between an asynchronous network-based model of Raft and ADORÉ. Full proofs can be found in the source code [10].

C.1 SRaft and ADORÉ

The most important part of the \mathbb{R} refinement relation is the $logMatch$ property (Fig. 17), which maps a replica's local log to its corresponding branch in the cache tree. The following proof sketch demonstrates the intuition for why this property is preserved by every operation.


$$\begin{aligned} \text{toLog}(tr, \text{nid}) &\triangleq \\ &\{C \in \text{activeBranch}(tr, \text{nid}) \mid C = \text{MCache}(_) \vee C = \text{RCache}(_)\} \\ \text{logMatch}(st_{\text{net}}, st_{\text{Adore}}) &\triangleq \\ \forall \text{nid}. \text{toLog}(\text{tree}(st_{\text{Adore}}), \text{nid}) &= \text{log}(\text{nodes}(st_{\text{net}})[\text{nid}]) \end{aligned}$$

Figure 17. The log-branch correspondence of \mathbb{R} .

$$\begin{aligned} \mathbb{R}_{\text{net}}(st, st'') &\triangleq \forall \text{nid}. \\ \text{log}(\text{nodes}(st)[\text{nid}]) &= \text{log}(\text{nodes}(st'')[\text{nid}]) \wedge \\ \text{time}(\text{nodes}(st)[\text{nid}]) &= \text{time}(\text{nodes}(st'')[\text{nid}]) \end{aligned}$$

Figure 18. The network-equivalence relation, \mathbb{R}_{net} .

Lemma C.1 (ADORE Refinement). *Suppose \mathbb{R} holds for some SRAFT state and a cache tree, tr , and that replica S 's local log is λ . For any valid SRAFT step where S 's new log is λ' there is a valid ADORE step to some tr' such that $\text{toLog}(tr', S) = \lambda'$.*

 [R_step (RefineNetAtomic.v)]

Proof. Consider each possible SRAFT operation. Neither `elect` nor `commit` change λ , so ADORE can take a `stutter` step and the result holds trivially. Both `invoke` and `reconfig` append a new method to λ , and the corresponding ADORE operations append an equivalent cache to the end of S 's active branch, preserving the relation. If `deliver`'s message is an election request or acknowledgement then λ does not change and likewise, `pull` only adds an `ECache` to the cache tree, which `toLog` ignores. The only other operation to change λ is a delivery of a commit request, in which case S takes the log λ' sent by some leader L . `push` adds a `CCache` to L 's branch whose supporters include S , which also makes it S 's active branch. Therefore, because `toLog` also ignores `CCaches`, we have $\text{toLog}(tr', S) = \text{toLog}(tr', L) = \lambda'$. \square

C.2 Raft and SRAFT


Recall that SRAFT is essentially the same as Raft, aside from some simplifying assumptions about how and when messages are delivered. To prove that Raft refines SRAFT, we must show that, despite these assumptions, SRAFT behaves equivalently to Raft, up to \mathbb{R}_{net} (Fig. 18), which states that the relevant parts of every replica's local state is the same. The following proof sketches explain the intuition for why each assumption is safe.

Definition C.2 (Valid Message). Upon receiving a message, every replica checks that it satisfies certain properties, and ignores it if it does not. Examples include a request with an insufficiently large timestamp, or an acknowledgement for a request that has already ended. A message is *valid* if it satisfies the properties and is therefore not ignored.

 [deliver_can_recv (NetworkPre.v)]

Lemma C.3 (SRAFT Valid Messages). *For any sequence of network events evs that results in a state st , there exists an*

sequence of events evs' that results in an equivalent state st' ($\mathbb{R}_{\text{net}}(st, st')$), such that evs' contains only valid messages.

 [E_inv (RefineNet.v)]

Proof. It is trivial to define evs' by simply filtering out invalid messages from evs . Since invalid messages are ignored anyway, this has no effect on the replicas' local states. \square


Definition C.4 (Ordered Messages). Messages m and m' are ordered if

$$(\text{time}(m), \text{vrsn}(m)) \leq_{\text{lex}} (\text{time}(m'), \text{vrsn}(m'))$$


where \leq_{lex} is the usual lexicographic order.

Definition C.5 (Locally Ordered Messages). A sequence of network events evs is locally ordered if for every m and m' such that $evs = \dots \cdot \text{deliver}(m) \cdot \dots \cdot \text{deliver}(m') \cdot \dots$ and $\text{to}(m) = \text{to}(m')$, m and m' are ordered.

Definition C.6 (Globally Ordered Message). A sequence of network events evs is globally ordered if for every m and m' such that $evs = \dots \cdot \text{deliver}(m) \cdot \dots \cdot \text{deliver}(m') \cdot \dots$, m and m' are ordered.

 [deliver_msg_ordered (NetworkOrdered.v)]


Lemma C.7 (SRAFT Globally Ordered). *For any sequence of network events evs that contains only valid messages that results in a state st , there exists a sequence of events evs' that results in an equivalent state st' ($\mathbb{R}_{\text{net}}(st, st')$), such that evs' is globally ordered.*

 [E_inv (RefineNetNoDup.v)]

Proof. We know evs is already locally ordered because non-locally ordered messages would be ignored, and there are no invalid messages by assumption. Therefore, all that remains is to show that sorting the globally unordered messages in evs does not affect the replicas' local states. Observe that receiving a message is a local operation in that it only affects the state of the recipient. Therefore, deliveries to different recipients are independent and can freely commute. Since we have established that messages are locally ordered, the only out-of-order messages must have different recipients, and can be sorted without affecting the final global state. \square

Definition C.8 (Atomic Deliveries). An operation in a sequence of network events evs is delivered atomically if every corresponding delivery (both of the request and the acknowledgements) is adjacent in evs .

Lemma C.9 (SRAFT Atomic). *For any sequence of network events evs that contains only globally ordered, valid messages that results in a state st , there exists a sequence of events evs' that results in an equivalent state st' ($\mathbb{R}_{\text{net}}(st, st')$), such that evs' has atomic deliveries.*

 [E_inv (RefineNetOrdered.v)]

 [E_inv (RefineNetAtomicReq.v)]

Proof. To construct evs' , for every operation, we must find all unadjacent corresponding deliveries and “push” them together in such a way that does not affect the resulting state. Because evs is globally ordered, and no leader uses the same timestamp-version number pair for its operations, any messages that come between two related deliveries must originate from a different leader. This also implies that the deliveries must have different recipients, because a replica would not accept two requests from different leaders with the same timestamp. Therefore, these delivery events can commute. By repeating this process, all delivery events can be rearranged so that corresponding ones are adjacent, and one can treat them as if they occurred atomically. \square

Lemma C.10 (Raft Refines SRaft). *For any sequence of network events evs that results in a state st , there exists an sequence of events evs' that results in an equivalent state st' ($\mathbb{R}_{\text{net}}(st, st')$), such that evs' contains only valid, globally ordered, and atomic messages.*

\clubsuit [E_inv (RefineLink.v)]

Proof. Trivial combination of Lemmas C.3, C.7 and C.9. \square

Theorem C.11 (Raft Refines ADORE). *Suppose \mathbb{R} holds for some Raft state and a cache tree, tr , and that replica S 's local log is λ . For any valid Raft step where S 's new log is λ' there is a valid ADORE step to some tr' such that $\text{toLog}(tr', S) = \lambda'$.*

\clubsuit [R_rado_network (RefineLink.v)]

Proof. Trivial combination of Lemmas C.1 and C.10. \square

D ADO and ADORE Formal Semantics

For reasons of clarity and space, Section 3 presented the semantics of the ADORE/CADO models piecemeal and omitted certain trivial cases. For completeness this section presents the full semantics of these models as well as the original ADO.

D.1 ADO

State. The ADO state has many elements in common with ADORE. The distributed system state (Σ_{ADO}) is modeled as a quadruple of the committed methods (*PersistLog*), the cache tree of uncommitted methods (*CacheTree*), a *CIDMap* that remembers every client's active cache, and an *OwnerMap* that remembers the unique leader at every timestamp. The tree is defined as a set of *Caches*, which use a *CID* data structure to induce the tree structure. A *CID* contains meta-data about when a *Cache* was added and by whom as well as a pointer to the parent. Ev_{ADO} defines all of the possible outcomes for pull, method invocation, and push. Every operation's behavior is divided into a generation rule that outputs an Ev_{ADO} , and an interpretation rule that consumes an Ev_{ADO} with $interp_{\text{ADO}}$ to construct Σ_{ADO} .

Log Generation. The pull oracle (\odot_{pull}) returns either *Ok* if the election succeeds, *Preempt* if the leader received

too few votes, but still took away another leader's supporters, or *Fail* if the election had no effect. The time chosen by the oracle must be larger than that of the parent cache and the *noOwnerAt* precondition guarantees that another leader has not already succeeded with the same timestamp.

Method invocation simply requires the caller's active cache to still be present in the cache tree. This both prevents methods from being called without first calling pull and stops replicas from continuing to use stale states after a different one was committed.

If the push oracle (\odot_{push}) decides that a commit succeeds it returns the *CID* of an uncommitted method that belongs to the caller and has the caller's current timestamp. Note that *nid* must be the *maxOwner*, which prevents leaders from being able to commit after having been preempted by a leader with a larger timestamp.

Log Interpretation. The failure events (Pull^- , Invoke^- , and Push^-) are all no-ops because they represent cases where network failures prevented enough messages from being delivered. A successful pull sets the caller's active cache and updates the *OwnerMap* with the new leader's time. It also blocks any leaders from being elected with an earlier time by setting them to *NoOwn*. In the event of a partially successful pull, only the times are updated.

Method invocation simply adds a new cache to the cache tree and updates the caller's active cache. A successful push partitions the tree into the committed caches (i.e., the ancestors of *ccid*) and the stale caches (i.e., all of the sibling branches). The committed branch is appended to the persistent log and the stale branches are discarded. Children of *ccid* remain in the tree because they could still be committed by a later push.

D.2 ADORE

State. The ADORE state is similar to the ADO model. The main differences are the caches store more information and there is no persistent log. Instead both committed and uncommitted caches are packed into the *CacheTree*. The tree also does away with the *CID* data type in favor of a simple unique numeric identifier for each cache. Whereas the ADO caches only stored methods, ADORE has four types of *Cache*: *ECache* (election), *MCache* (method), *RCache* (reconfiguration) and *CCache* (commit). This additional information means there is no need for *CIDMap* or *OwnerMap* because they can be computed directly from the tree. The *TimeMap* remembers each replica's largest observed timestamp.

Semantics. Like the ADO model, pull and push use oracles (\odot_{pull} and \odot_{push}) to nondeterministically determine their outcomes and all operations can fail at any time. On success, \odot_{pull} chooses a set of supporters from the configuration and a time that is larger than they have observed. There is no need for a separate *Pull** outcome because Q may or may

$$\begin{aligned}
CID &\triangleq \langle \mathbb{N}_{nid} * \mathbb{N}_{time} * CID \rangle \mid \mathbf{Root} \\
Cache &\triangleq CID * Method \\
PersistLog &\triangleq List(Cache) \\
CacheTree &\triangleq Set(Cache) \\
CIDMap &\triangleq \mathbb{N}_{nid} \rightarrow CID \\
OwnerMap &\triangleq \mathbb{N}_{time} \rightarrow (\mathbb{N}_{nid} \mid \mathbf{NoOwn}) \\
\Sigma_{ADO} &\triangleq PersistLog * CacheTree * CIDMap * OwnerMap \\
Ev_{ADO} &\triangleq Pull^+(\mathbb{N}_{nid} * \mathbb{N}_{time} * CID) \\
&\quad \mid Pull^*(\mathbb{N}_{nid} * \mathbb{N}_{time}) \\
&\quad \mid Pull^-(\mathbb{N}_{nid}) \\
&\quad \mid Invoke^+(\mathbb{N}_{nid} * Method) \\
&\quad \mid Invoke^-(\mathbb{N}_{nid}) \\
&\quad \mid Push^+(\mathbb{N}_{nid} * CID) \\
&\quad \mid Push^-(\mathbb{N}_{nid}) \\
interp_{ADO} &: Ev_{ADO} \rightarrow \Sigma_{ADO} \rightarrow \Sigma_{ADO} \\
interpAll_{ADO}(evs) &\triangleq fold(evs, interp_{ADO}, initState) \\
Op &\triangleq pull : \mathbb{N}_{nid} \rightarrow List(Ev_{ADO}) \rightarrow List(Ev_{ADO}) \\
&\quad \mid invoke : \mathbb{N}_{nid} \rightarrow Method \rightarrow List(Ev_{ADO}) \rightarrow List(Ev_{ADO}) \\
&\quad \mid push : \mathbb{N}_{nid} \rightarrow List(Ev_{ADO}) \rightarrow List(Ev_{ADO})
\end{aligned}$$

Figure 19. ADO state, events, and operations.

$$\begin{aligned}
\circlearrowleft_{pull} &: List(Ev_{ADO}) \rightarrow \mathbb{N}_{nid} \rightarrow (Ok(\mathbb{N}_{time} * CID) \mid Preempt(\mathbb{N}_{time}) \mid Fail) \\
\circlearrowleft_{push} &: List(Ev_{ADO}) \rightarrow \mathbb{N}_{nid} \rightarrow (Ok(CID) \mid Fail) \\
&\frac{\text{VALIDPULLORACLE} \quad timeOf(cid) < time \quad noOwnerAt(evs, time) \quad (cid \in caches(evs) \vee cid = root(evs))}{\circlearrowleft_{pull}(evs, nid) = Ok(time, cid)} \\
&\frac{\text{VALIDPULLORACLEPARTIAL} \quad time \notin dom(owners(evs))}{\circlearrowleft_{pull}(evs, nid) = Preempt(time)} \\
&\frac{\text{VALIDPUSHORACLE} \quad nidOf(ccid) = maxOwner(evs) = nid \quad timeOf(ccid) = timeOf(cids(evs)[nid]) \quad ccid \in caches(evs)}{\circlearrowleft_{push}(evs, nid) = Ok(cid)}
\end{aligned}$$

Figure 20. ADO valid oracle rules.

not be a quorum. As in the ADO model, failure events are no-ops. A successful pull updates the timestamps of the supporters and, if the supporters constitute a quorum, adds a new *ECache*. The parent of the *ECache* is the largest cache supported by any of the supporters.

If a method invocation succeeds then the caller's current timestamp must equal the time of C_A , the caller's active cache. This prevents a leader from continuing to call methods after learning of a newer leader. Method invocation simply adds

an *MCache* as a child of the caller's active cache. The version number is increased by 1 to help make $>$ a total order.

`reconfig` behaves like a method invocation, except that it replaces the current configuration and must satisfy R1⁺, R2, and R3. These conditions restrict how much the configurations can diverge before one is committed.

Finally, similarly to pull, a successful push updates the supporters' times and adds a *CCache* if there is a quorum.

$$\begin{array}{c}
\text{PULLSUCCESS} \\
\frac{\mathbb{O}_{\text{pull}}(evs, nid) = Ok(time, cid)}{\mathbb{O} \vdash \text{pull}(nid) : evs \rightsquigarrow evs \bullet \text{Pull}^+(nid, time, cid)} \\
\\
\text{PULLFAILURE} \\
\frac{\mathbb{O}_{\text{pull}}(evs, nid) = Fail}{\mathbb{O} \vdash \text{pull}(nid) : evs \rightsquigarrow evs \bullet \text{Pull}^-(nid)} \\
\\
\text{METHODFAILURE} \\
\frac{cids(ev)[nid] \notin caches(ev)}{\mathbb{O} \vdash \text{invoke}(nid, M) : evs \rightsquigarrow evs \bullet \text{Inooke}^-(nid)} \\
\\
\text{PULLPREEMPT} \\
\frac{\mathbb{O}_{\text{pull}}(evs, nid) = Preempt(time)}{\mathbb{O} \vdash \text{pull}(nid) : evs \rightsquigarrow evs \bullet \text{Pull}^*(nid, time)} \\
\\
\text{METHODINVOCATION} \\
\frac{cids(ev)[nid] \in caches(ev)}{\mathbb{O} \vdash \text{invoke}(nid, M) : evs \rightsquigarrow evs \bullet \text{Inooke}^+(nid, M)} \\
\\
\text{PUSHSUCCESS} \\
\frac{\mathbb{O}_{\text{push}}(evs, nid) = Ok(ccid)}{\mathbb{O} \vdash \text{push}(nid) : evs \rightsquigarrow evs \bullet \text{Push}^+(nid, ccid)} \\
\\
\text{PUSHFAILURE} \\
\frac{\mathbb{O}_{\text{push}}(evs, nid) = Fail}{\mathbb{O} \vdash \text{push}(nid) : evs \rightsquigarrow evs \bullet \text{Push}^-(nid)}
\end{array}$$

Figure 21. ADO event generation rules.

$$\begin{array}{c}
\text{INTERPPULLSUCCESS} \\
\frac{cids' = cids[nid \mapsto \langle nid, time, cid \rangle] \quad owns' = voteNoOwn(owns[time \mapsto nid], time - 1)}{interp_{\text{ADO}}(\text{Pull}^+(nid, time, cid), (p, cs, cids, owns)) = (p, cs, cids', owns')} \\
\\
\text{INTERPPULLPREEMPT} \\
\frac{owns' = voteNoOwn(owns, time)}{interp_{\text{ADO}}(\text{Pull}^*(nid, time), (p, cs, cids, owns)) = (p, cs, cids, owns')} \\
\\
\text{INTERPPULLFAILURE} \\
\frac{}{interp_{\text{ADO}}(\text{Pull}^-(nid), (p, cs, cids, owns)) = (p, cs, cids, owns)} \\
\\
\text{INTERPMETHODINVOCATION} \\
\frac{cs' = cs \cup \{cids[nid], M\} \quad cids' = cids[nid \mapsto nextCID(cid)]}{interp_{\text{ADO}}(\text{Inooke}^+(nid, M), (p, cs, cids, owns)) = (p, cs', cids', owns)} \\
\\
\text{INTERPMETHODFAILURE} \\
\frac{}{interp_{\text{ADO}}(\text{Inooke}^-(nid), (p, cs, cids, owns)) = (p, cs, cids, owns)} \\
\\
\text{INTERPPUSHSUCCESS} \\
\frac{(\vec{c}_{ok}, cs') = partition(cs, ccid)}{interp_{\text{ADO}}(\text{Push}^+(nid, ccid), (p, cs, cids, owns)) = (p \bullet \vec{c}_{ok}, cs', cid, owns)} \\
\\
\text{INTERPPUSHFAILURE} \\
\frac{}{interp_{\text{ADO}}(\text{Push}^-(nid), (p, cs, cids, owns)) = (p, cs, cids, owns)}
\end{array}$$

Figure 22. ADO event interpretation rules.

However, instead of creating a new leaf node, the *CCache* is inserted between the selected *MCache* and its children. The *CCache* copies its parent's timestamp and version number, but $>$ is defined such that it is still greater than its parent. A successful push chooses a set of supporters and an arbitrary

uncommitted *MCache* or *RCache* belonging to the caller. The cache must also have a timestamp at least as large as any observed by the supporters. This prevents replicas from committing commands that come from anyone other than the most recent leader.

$$\begin{aligned}
nidOf(cid) &\triangleq \mathbf{let} \langle nid, _ \rangle = cid \mathbf{in} \, nid \\
timeOf(cid) &\triangleq \mathbf{let} \langle _, time, _ \rangle = cid \mathbf{in} \, time \\
root(eps) &\triangleq \mathbf{let} (p, _ _, _) = interpAll_{ADO}(eps) \mathbf{in} \, \mathbf{if} \, p \neq [] \mathbf{then} \, last(p) \mathbf{else} \, \mathbf{Root} \\
caches(eps) &\triangleq \mathbf{let} (_ _, cs, _ _) = interpAll_{ADO}(eps) \mathbf{in} \, cs \\
cids(eps) &\triangleq \mathbf{let} (_ _, cids, _) = interpAll_{ADO}(eps) \mathbf{in} \, cids \\
owners(eps) &\triangleq \mathbf{let} (_ _, _ _, owns) = interpAll_{ADO}(eps) \mathbf{in} \, owns \\
noOwnerAt(eps, time) &\triangleq time \notin dom(owners(eps)) \vee owners(eps)[time] = \mathbf{NoOwn} \\
maxOwner(eps) &\triangleq \mathbf{let} \, owns = owners(eps) \mathbf{in} \, owns[\max(dom(owns))] \\
cid_1 < cid_2 &\triangleq cid_2 \neq \mathbf{Root} \wedge \mathbf{let} \langle _ _, parent \rangle = cid_2 \mathbf{in} \, cid_1 = parent \vee cid_1 < parent \\
cid_1 \leq cid_2 &\triangleq cid_1 < cid_2 \vee cid_1 = cid_2 \\
voteNoOwn(owns, time) &\triangleq owns[t \mapsto \mathbf{NoOwn} \mid \forall t \leq time. t \notin dom(owns)] \\
nextCID(cid) &\triangleq \mathbf{let} \langle nid, time, _ \rangle = cid \mathbf{in} \, \langle nid, time, cid \rangle \\
partition(cs, cid) &\triangleq \mathbf{let} \vec{c}_{ok} = sort(\{(c, M) \in cs \mid c \leq cid\}) \mathbf{in} \\
&\quad \mathbf{let} \, cs' = \{(c, M) \in cs \mid cid < c\} \mathbf{in} \, (\vec{c}_{ok}, cs')
\end{aligned}$$

Figure 23. ADO auxiliary functions.

$$\begin{aligned}
Cache &\triangleq ECache(\mathbb{N}_{nid} * \mathbb{N}_{time} * \mathbb{N}_{vrsn} * Set(\mathbb{N}_{nid}) * Config) \\
&\quad | MCache(\mathbb{N}_{nid} * \mathbb{N}_{time} * \mathbb{N}_{vrsn} * Method * Config) \\
&\quad | CCache(\mathbb{N}_{nid} * \mathbb{N}_{time} * \mathbb{N}_{vrsn} * Set(\mathbb{N}_{nid}) * Config) \\
&\quad | RCache(\mathbb{N}_{nid} * \mathbb{N}_{time} * \mathbb{N}_{vrsn} * Config) \\
CacheTree &\triangleq \mathbb{N}_{cid} \rightarrow \mathbb{N}_{cid} * Cache \\
TimeMap &\triangleq \mathbb{N}_{nid} \rightarrow \mathbb{N}_{time} \\
\Sigma_{Adore} &\triangleq CacheTree * TimeMap \\
Op &\triangleq \mathbf{pull} : \mathbb{N}_{nid} \rightarrow \Sigma_{Adore} \rightarrow \Sigma_{Adore} \\
&\quad | \mathbf{invoke} : \mathbb{N}_{nid} \rightarrow Method \rightarrow \Sigma_{Adore} \rightarrow \Sigma_{Adore} \\
&\quad | \mathbf{reconfig} : \mathbb{N}_{nid} \rightarrow Config \rightarrow \Sigma_{Adore} \rightarrow \Sigma_{Adore} \\
&\quad | \mathbf{push} : \mathbb{N}_{nid} \rightarrow \Sigma_{Adore} \rightarrow \Sigma_{Adore}
\end{aligned}$$

Figure 24. ADORE state.

Parameters

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

Assumptions about R1^+ and isQuorum (REFLEXIVE) $\text{R1}^+(cf, cf)$ (OVERLAP) $\text{R1}^+(cf, cf') \wedge \text{isQuorum}(Q, cf) \wedge \text{isQuorum}(Q', cf') \implies Q \cap Q' \neq \emptyset$ **Definitions**

$$\text{R2}(tr, C) \triangleq \forall C' \in tr. C' = \text{RCache}(_) \wedge C' \uparrow C \implies \exists C'' \in tr. C'' = \text{CCache}(_) \wedge C' \uparrow C'' \wedge C'' \uparrow C$$

$$\text{R3}(tr, C) \triangleq \exists C' \in tr. C' = \text{CCache}(_) \wedge \text{time}(C') = \text{time}(C) \wedge C' \uparrow C$$

$$\text{canReconf}(tr, C, ncf) \triangleq \text{R1}^+(\text{conf}(C), ncf) \wedge \text{R2}(tr, C) \wedge \text{R3}(tr, C)$$

Figure 25. Configuration/quorum parameters and definitions.

$$C_1 > C_2 \triangleq (\text{time}(C_1), \text{vrsn}(C_1)) > (\text{time}(C_2), \text{vrsn}(C_2))$$

$$\vee (\text{time}(C_1), \text{vrsn}(C_1)) = (\text{time}(C_2), \text{vrsn}(C_2)) \wedge C_1 = \text{CCache}(_) \wedge C_2 \neq \text{CCache}(_)$$

$$\text{freshCID}(tr) \triangleq \max \{ \text{cid}(C) \mid C \in tr \} + 1$$

$$\text{addLeaf}(st, C_p, C_{new}) \triangleq (\text{tree}(st)[\text{freshCID}(\text{tree}(st)) \mapsto (C_p, C_{new})], \text{times}(st))$$

$$\text{insertBtw}(st, C_p, C_{new}) \triangleq \text{let } tr' = \text{tree}(st)[\text{cid}(C) \mapsto (\text{cid}(C_{new}), C) \mid \forall (_, C) \in \text{tree}(st)] \text{ in } \\ (tr'[\text{freshCID}(\text{tree}(st)) \mapsto (C_p, C_{new})], \text{times}(st))$$

$$C \uparrow C' \triangleq C = \text{parent}(C') \vee C \uparrow \text{parent}(C')$$

$$\text{setTimes}(st, Q, t) \triangleq (\text{tree}(st), \text{times}(st)[s \mapsto t \mid \forall s \in Q])$$

$$\text{validSupp}(nid, Q, C) \triangleq nid \in Q \wedge Q \subseteq \text{mbrs}(\text{conf}(C))$$

$$\text{isLeader}(st, nid, t) \triangleq \text{times}(st)[nid] = t$$

$$\text{mostRecent}(tr, Q) \triangleq \max_{>} \{ C \in tr \mid Q \cap \text{supporters}(C) \neq \emptyset \}$$

$$\text{activeCache}(tr, nid) \triangleq \max_{>} \{ C \in tr \mid \text{caller}(C) = nid \}$$

$$\text{lastCommit}(tr, nid) \triangleq \max_{>} \{ C \in tr \mid nid \in \text{supporters}(C) \wedge C = \text{CCache}(_) \}$$

$$\text{canCommit}(C, nid, st) \triangleq (C = \text{MCache}(_) \vee C = \text{RCache}(_)) \wedge \text{caller}(C) = nid \\ \wedge \text{isLeader}(st, nid, \text{time}(C)) \wedge C > \text{lastCommit}(\text{tree}(st), nid)$$

Figure 26. ADORE auxiliary definitions.

$$\begin{array}{l}
\mathbb{O}_{pull} : \Sigma_{Adore} \rightarrow \mathbb{N}_{nid} \rightarrow (Ok(Set(\mathbb{N}_{nid}) * \mathbb{B} * Cache * \mathbb{N}_{time}) | Fail) \\
\mathbb{O}_{push} : \Sigma_{Adore} \rightarrow \mathbb{N}_{nid} \rightarrow (Ok(Set(\mathbb{N}_{nid}) * \mathbb{B} * Cache) | Fail) \\
\text{VALIDPULLORACLE} \\
\frac{Q_{ok} \triangleq isQuorum(Q, conf(C_{max})) \quad \text{validSupp}(nid, Q, C_{max}) \quad \forall s \in Q. times(st)[s] < t \quad C_{max} \triangleq mostRecent(tree(st), Q)}{\mathbb{O}_{pull}(st, nid) = Ok(Q, Q_{ok}, C_{max}, t)} \\
\text{VALIDPUSHORACLE} \\
\frac{Q_{ok} \triangleq isQuorum(Q, conf(C_M)) \quad \text{validSupp}(nid, Q, C_M) \quad \forall s \in Q. times(st)[s] \leq time(C_M) \quad canCommit(C_M, nid, st)}{\mathbb{O}_{push}(st, nid) = Ok(Q, Q_{ok}, C_M)}
\end{array}$$

Figure 27. ADORE valid oracle rules.

$$\begin{array}{l}
\text{PULLOK} \\
\frac{\mathbb{O}_{pull}(st, nid) = Ok(Q, Q_{ok}, C_{max}, t) \quad st' \triangleq setTime(s, Q, t) \quad C_{new} \triangleq ECache(nid, t, 0, Q, conf(C_{max}))}{\mathbb{O} \vdash \text{pull}(nid) : st \rightsquigarrow \text{if } Q_{ok} \text{ then } addLeaf(st', C_{max}, C_{new}) \text{ else } st'} \\
\text{INVOKEOK} \\
\frac{C_A \triangleq activeCache(tree(st), nid) \quad isLeader(st, nid, time(C_A)) \quad C_{new} \triangleq MCache(nid, time(C_A), vrsn(C_A) + 1, M, conf(C_A))}{\mathbb{O} \vdash \text{invoke}(nid, M) : st \rightsquigarrow addLeaf(st, C_A, C_{new})} \\
\text{RECONFIGOK} \\
\frac{C_A \triangleq activeCache(tree(st), nid) \quad isLeader(st, nid, time(C_A)) \quad canReconf(tree(st), C_A, ncf) \quad C_{new} \triangleq RCache(nid, time(C_A), vrsn(C_A) + 1, ncf)}{\mathbb{O} \vdash \text{reconfig}(nid, ncf) : st \rightsquigarrow addLeaf(st, C_A, C_{new})} \\
\text{PUSHOK} \\
\frac{st' \triangleq setTime(st, Q, time(C_M)) \quad \mathbb{O}_{push}(st, nid) = Ok(Q, Q_{ok}, C_M) \quad C_{new} \triangleq CCache(nid, time(C_M), vrsn(C_M), Q, conf(C_M))}{\mathbb{O} \vdash \text{push}(nid) : st \rightsquigarrow \text{if } Q_{ok} \text{ then } insertBtw(st', C_M, C_{new}) \text{ else } st'} \\
\text{PULLNOOP} \\
\frac{\mathbb{O}_{pull}(st, nid) = Fail}{\mathbb{O} \vdash \text{pull}(nid) : st \rightsquigarrow st} \\
\text{INVOKENOOP} \\
\frac{}{\mathbb{O} \vdash \text{invoke}(nid, M) : st \rightsquigarrow st} \\
\text{RECONFIGNOOP} \\
\frac{}{\mathbb{O} \vdash \text{reconfig}(nid, ncf) : st \rightsquigarrow st} \\
\text{PUSHNOOP} \\
\frac{\mathbb{O}_{push}(st, nid) = Fail}{\mathbb{O} \vdash \text{push}(nid) : st \rightsquigarrow st}
\end{array}$$

Figure 28. Semantics of ADORE operations.