Bolt-On Strong Consistency: Specification, Implementation, and Verification

NICHOLAS V. LEWCHENKO, University of Colorado Boulder, USA
GOWTHAM KAKI, University of Colorado Boulder, USA
BOR-YUH EVAN CHANG*, University of Colorado Boulder, USA and Amazon, USA

Strongly-consistent replicated data stores are a popular foundation for many kinds of online services, but their implementations are very complex. Strong replication is not *available* under network partitions, and so achieving a functional degree of fault-tolerance requires correctly implementing *consensus algorithms* like Raft and Paxos. These algorithms are notoriously difficult to reason about, and many data stores implement custom variations to support unique performance tradeoffs, presenting an opportunity for automated verification tools. Unfortunately, existing tools that have been applied to distributed consensus demand too much developer effort, a problem stemming from the low-level programming model in which consensus and strong replication are implemented—asynchronous message passing—which thwarts decidable automation by exposing the details of asynchronous communication.

In this paper, we consider the implementation and automated verification of strong replication systems *as applications* of weak replicated data stores. Weak stores, being available under partition, are a suitable foundation for performant distributed applications. Crucially, they abstract asynchronous communication and allow us to derive local-scope conditions for the verification of consensus safety. To evaluate this approach, we have developed a verified-programming framework for the weak replicated state model, called Super-V. This framework enables SMT-based verification based on local-scope artifacts called *stable update preconditions*, replacing standard-practice global inductive invariants. We have used our approach to implement and verify a strong replication system based on an adaptation of the Raft consensus algorithm.

CCS Concepts: • **Software and its engineering** → *Formal software verification*; Distributed programming languages; • **Networks** → *Protocol testing and verification.*

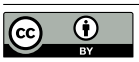
ACM Reference Format:
Nicholas V. Lewchenko, Gowtham Kaki, and Bor-Yuh Evan Chang. 2025. Bolt-On Strong Consistency: Specification, Implementation, and Verification. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 137 (April 2025), 28 pages. https://doi.org/10.1145/3720502

1 Introduction

Replicated state is a common abstraction employed by distributed applications in which each application node operates on the local replica of the application state while also propagating the updates to the remote replicas. The *Replicated State Machine* model studied widely in the distributed systems literature [Ailijiang et al. 2019; Lynch 1996; Schneider 1990] adopts *strongly-consistent* replication, where updates are applied uniformly at all replicas in a linear order consistent with their temporal order, i.e., their execution is *linearizable* [Herlihy and Wing 1990]. This property simplifies

*Bor-Yuh Evan Chang holds concurrent appointments at the University of Colorado Boulder and as an Amazon Scholar. This paper describes work performed at the University of Colorado Boulder and is not associated with Amazon.

Authors' Contact Information: Nicholas V. Lewchenko, University of Colorado Boulder, Boulder, USA, nicholas.lewchenko@colorado.edu; Gowtham Kaki, University of Colorado Boulder, Boulder, USA, gowtham.kaki@colorado.edu; Bor-Yuh Evan Chang, University of Colorado Boulder, Boulder, USA and Amazon, Seattle, USA, evan.chang@colorado.edu.



ACM 2475-1421/2025/4-ART137
https://doi.org/10.1145/3720502

application programming by obviating the need to reason about concurrency. However, it comes at the inevitable cost of increased latency, because preempting concurrency requires system-wide coordination, which cannot be achieved within a bounded time on an unreliable network [Fischer et al. 1985]. Furthermore, strong replication systems are *unavailable* under network partitions: nodes must be blocked from making progress when network failures divide the system into disjoint subnets [Gilbert and Lynch 2002].

The high cost of strong replication is unacceptable in many real-world distributed applications that prioritize low latency and high availability. Such applications have come to adopt *weakly-consistent* replication, which never blocks nodes from performing updates to their local replicas. As a consequence, updates must be allowed to apply on remote replicas in different orders, possibly violating linearizability. Despite this, weak replication has found use in a variety of distributed applications ranging from e-commerce [DeCandia et al. 2007; Martyanov 2020] to collaborative editors [Teixeira 2017; Wallace 2019], which establish weaker guarantees, such as *convergence* and *no lost writes*. Several mostly-automated verification frameworks for these weaker properties have been proposed in the last decade [Liu et al. 2020; Nagar and Jagannathan 2019; Zakhour et al. 2023].

Compared to weak replication, the strong replicated state model makes reasoning about applications easy: the application state behaves as a centralized database interface—a crucial property for applications that maintain finite resources and must prevent double-spending bugs. *Implementing* strong consistency efficiently, on the other hand, is extremely complex as it requires a careful design that tolerates as many node and network failures as possible while orchestrating system-wide *consensus* for each replicated operation. Researchers have proposed several fault-tolerant *consensus algorithms* as the basis of strong replication implementations. These include the well-known Paxos [Lamport 1998] and Raft [Ongaro and Ousterhout 2014] algorithms, as well as variants such as Fast Paxos [Lamport 2006], Egalitarian Paxos [Moraru et al. 2013] and Flexible Paxos [Howard et al. 2017]. While these provide a general solution, most distributed applications make unique design choices that benefit from more specific performance and fault-tolerance optimizations. For this reason, commodity distributed data stores tend to design and implement their own consensus algorithm variations for their own specific use cases [Confluent 2024; Corbett et al. 2013; Grieger 2019; Junqueira et al. 2011; Schultz et al. 2021]. Regardless of their specific design choices, the internal logic of these protocols is invariably complex to account for the possibility of failures at each step of the execution, which makes reasoning about their correctness notoriously difficult.

To address the aforementioned problems, formal verification for distributed consensus algorithms emerged as an active area of research. Several domain-specific languages and integrated verification frameworks for distributed programming have emerged in the last decade [Kragl et al. 2020; Padon et al. 2016; Sergey et al. 2017; Taube et al. 2018; Wilcox et al. 2015]. While they differ in their choice of language-level abstractions and automation techniques, they adopt the same fundamental approach to protocol specification and verification: inductive invariants on a global transition system that represents a network of nodes and message queues. In practice, these invariants must exhaustively detail the subtle relationships between elements of the system, such as the content of messages in a queue and the value of their sender's local state. Such invariants are hard to identify [Padon et al. 2016] and have complex syntactic structure not amenable to automated reasoning [Padon et al. 2017; v. Gleissenthall et al. 2019]. Despite such drawbacks, the low-level approach to verification persists as protocols that orchestrate strong consistency are almost always implemented on the low-level message-passing programming model.

Abstraction is the key to improving these implementation and verification practices. Weak replication simplifies the distributed programming model by hiding the explicit mechanism of communication between nodes, and yet it remains efficient for the implementation of general distributed applications. Would it be to our benefit to implement strong replication systems as

applications of weak replicated stores, i.e., *bolt on* strong consistency? The compositional approach to building replicated state abstractions is already a widely adopted practice. Conflict-free Replicated Data Types (CRDTs) [Shapiro et al. 2011], which bring lock-free data structures to replicated state, are often implemented on top of weak replicated stores that offer *causal delivery*, strengthening that guarantee to *strong eventual consistency* [Brown et al. 2014; De Porre et al. 2023; Liu et al. 2020; Nagar and Jagannathan 2019; Sivaramakrishnan et al. 2015; Zakhour et al. 2023]. Bailis et al. [2013] further demonstrate that causal delivery itself can be implemented efficiently as a bolt-on shim layer to a weaker replicated store, and Redmond et al. [2023] explores the verification of causal delivery mechanisms. Why should the implementation of consensus algorithms for strong replication forgo all these developments, returning to the full complexity of asynchronous message-passing?

In this paper, we extend the compositional approach to building replicated state abstractions from weak consistency to strong consistency. In particular, we re-consider the underlying consensus problem from first principles, with a weak, causally-consistent replication system as our underlying model instead of asynchronous message-passing. The benefits of doing so, as we demonstrate, are manifold. First, the replicated state model allows us to break the consensus verification problem into two simpler proof obligations: *monotonicity* and *commutativity*. This decomposition is based on the intuition that, in the context of a consensus protocol, if all replicas of the protocol state (which contains within it the decision value) eventually converge, and a decision once made by any replica cannot be reverted (i.e., it is monotonic), then it is inevitable that replicas will only ever make decisions that agree, which is precisely the definition of consensus.

The second advantage of the replicated state system model is that it abstracts away the complexity of the underlying network layer, resulting in simpler verification conditions that support decidable automation. In particular, abstracting away the network layer eliminates the need to reason about individual messages and message buffers, complex syntactic structures which often lead to undecidable quantifier alternation [v. Gleissenthall et al. 2019]. This is the same intuition leveraged by several verification frameworks to make verification of CRDTs tractable in practice [De Porre et al. 2023; Liu et al. 2020; Zakhour et al. 2023]. Our approach extends this leverage to consensus algorithms, for which we develop a new proof theory based on *stable update preconditions* instead of inductive invariants. Notably, our proof theory's verification conditions do not introduce higher-order structures or quantifier alternations that could cause undecidability (Sec. 4). Decidability of overall verification, however, depends on the domain-specific implementation choices made by the programmer.

Finally, we observe that a causally-consistent system model allows us to simplify the design of consensus algorithms. This is for the same reason that causal broadcast libraries simplify many CRDT implementations [Redmond et al. 2023]: explicit, application-level dependency tracking can be factored out when the underlying store provides it. This allowed us to eliminate some of the complexity in our implemented variant of Raft [Ongaro and Ousterhout 2014], by relying on the fact that updates are delivered in causal order. Moreover, we were able to safely incorporate a non-standard optimization—the option to use minority/super-majority quorums [Howard et al. 2017]—with little additional effort. The adaptation to the replicated state model is otherwise straightforward: message sends are replaced with updates to the shared state and receives are simply reads. We discuss the design of this adapted algorithm, which we call *Ferry*, in Section 5.

We implemented our proof theory in a verifiable programming framework called Super-V. Our framework provides a Haskell DSL for implementing distributed applications, which allows them to execute atop a casually-consistent weak replicated store—a high-level abstraction familiar to distributed application developers. Our framework also compiles these implementations to an SMT-LIB representation and generates verification conditions for deciding their safety. We used this framework to implement, evaluate, and verify a strongly-consistent replicated store based

on our Ferry consensus algorithm. Despite being implemented at a high level of abstraction, this replicated store is comparable in performance—and scalable to the same network sizes—to stores implemented in standard practice. To the best of our knowledge, this Ferry-based store is the first verified strong consistency implementation that leverages the flexible quorums optimization first proposed in Flexible Paxos [Howard et al. 2017].

Contributions. To summarize, we present the following contributions in this paper:

- The decomposition of global consensus safety, in the weak replication model, into *monotonicity* and *commutativity* of updates—properties that can be verified in local terms.
- A novel proof theory for verifying consensus protocols in the weak replication model that eschews system-wide inductive invariants in favor of more tractable *stable update preconditions*. The proof theory is supplemented with a precise characterization of the resulting verification conditions and their decidability.
- Super-V, a programming framework that implements our proof theory to enable the implementation of weak replication applications that are both executable and verifiable.
- A verified strong replicated data store based on *Ferry*: a consensus algorithm that is extensionally identical to Raft but with simpler internal logic made possible by leveraging the weak replication programming model. We present an extensive empirical evaluation comparing Ferry against against a Raft-based strong replication system.

2 Motivation

In this section, we motivate our approach using a classic distributed consensus problem: *leader election*. We will illustrate the difficulties that arise in the traditional verification approach, and show how they can be avoided when using weak replication as the programming model.

2.1 Traditional Verification: Global Inductive Invariant

The goal of a leader election algorithm is to establish a single *leader* among a network of nodes, which all nodes agree upon. Fig. 1a defines a toy *one-shot* election algorithm, using a message-passing programming model typical for distributed algorithms. Any node can execute the `castVote`(N_c) action at any time—unless its local state (the `local` variable) says it has already voted, the node sends a `VoteFrom`(N_v) message (where N_v is the sender's unique `self` ID) to the candidate node N_c. When the candidate N_c receives `VoteFrom`(N_v), it executes the appropriate `handle` procedure, which adds the voter N_v to N_c's local `myVotes` set. When a node N_c receives a majority number of votes, it marks itself as leader and broadcasts a `Leader`(N_c) that instructs peers to do the same.

This algorithm is simplistic: nodes vote only once, and so it cannot recover from a split vote and has no liveness guarantee. However, it does provide a *consensus safety* guarantee: it ensures that two different leaders $N_1 \neq N_2$ cannot both be recognized. To be more specific, this guarantee holds for two dimensions of the execution. Across *time*, a node cannot satisfy `isLeader`(N_1) now and then satisfy `isLeader`(N_2) later. Across *space*, one node cannot satisfy `isLeader`(N_1) while a distant node simultaneously satisfies `isLeader`(N_2)[1].

Existing work has focused exclusively on verifying the space dimension of consensus safety [Hawblitzel et al. 2015; Padon et al. 2017; Taube et al. 2018; v. Gleissenthall et al. 2019; Woos et al. 2016], likely because it is the least trivial aspect. The standard approach models the entire distributed network as a single transition system, and defines (space) consensus safety as an invariant on the

[1] In Lamport's writings on Paxos, the time and space aspects are called "stability" and "consistency," respectively [Lamport 2004]. This meaning of "consistency" is distinct from the *weak* and *strong consistency* we discussed in Sec. 1.

```
castVote(n) :=
    if ¬local.hasVoted then
        local.hasVoted := True;
        send VoteFrom(self) to n;

isLeader(n) := return(local.leader = n);

handle VoteFrom(n) :=
    local.myVotes.insert(n);
    if quorum(local.myVotes) then
        local.leader := self;
        send Leader(self) to All;
handle Leader(n) :=
    local.leader := n;
```

(a) General Message Passing.

```
castVote(n) :=
    if ¬local.votes(self) ≠ ⊥ then
        update Vote(self, n);

isLeader(n) :=
    let q := local.voters.filter(
        λn1. local.votes[n1] = n
    );
    return(isQuorum(q));

handle Vote(n1, n2) :=
    local.votes.insert(n1, n2);
```

(b) Weakly Replicated State.

Fig. 1. One-shot leader election algorithm implemented under two programming models. Each node in the network can arbitrarily execute the actions (castVote, isLeader), while the **handle** procedures automatically execute upon receipt of a message or update.

global network states. The following invariant I_{cs} is an example, for our toy election algorithm:

$$\Sigma \in I_{\mathsf{cs}} \quad \triangleq \quad \forall N_1, N_2, N_3, N_4. \quad \Sigma[N_1].\mathtt{isLeader}(N_2) \ \wedge \ \Sigma[N_3].\mathtt{isLeader}(N_4) \implies N_2 = N_4.$$

Here, Σ is a mapping from node IDs to node states, representing part of the global state of the network, and $\Sigma[N_1].\mathtt{isLeader}(N_2)$ denotes the output of $\mathtt{isLeader}(N_2)$ under local state $\Sigma[N_1]$.

The standard approach proceeds with an *induction* proof, showing that each network step takes every $\Sigma_1 \in I_{\mathsf{cs}}$ to another $\Sigma_2 \in I_{\mathsf{cs}}$. Unfortunately, the goal invariant is usually not inductive on its own, due to asynchronicity: in addition to the local states of nodes, the network state contains messages-in-transit which are not constrained by I_{cs}. This under-specification allows impossible messages—such as two $\mathtt{VoteFor}(N_v)$ messages sent from the same node to two different recipients—to appear "out of thin air" and break I_{cs}. Achieving induction requires us to strengthen the invariant with the following additional clauses. In addition to Σ, these clauses constrain Δ: a mapping from N to the set of messages that have so-far been sent to N (either delivered or still in-flight).

$$\forall N_1, N_2. \quad \Sigma[N_1].\mathtt{isLeader}(N_2) \implies \mathtt{Leader}(N_2) \in \Delta[N_1] \tag{1}$$

$$\forall N_1, N_2. \quad \mathtt{Leader}(N_1) \in \Delta[N_2] \implies \mathtt{isQuorum}(\Sigma[N_1].\mathtt{myVotes}) \tag{2}$$

$$\forall N_1, N_2. \quad N_1 \in \Sigma[N_2].\mathtt{myVotes} \implies \mathtt{VoteFrom}(N_1) \in \Delta[N_2] \tag{3}$$

$$\forall N_1, N_2. \quad \mathtt{VoteFrom}(N_1) \in \Delta[N_2] \implies \Sigma[N_1].\mathtt{hasVoted} \tag{4}$$

$$\forall N_1, N_2, N_3. \quad \mathtt{VoteFrom}(N_1) \in \Delta[N_2] \wedge \mathtt{VoteFrom}(N_1) \in \Delta[N_3] \implies N_2 = N_3 \tag{5}$$

The new clauses respectively assert that (1) a node's myLeader is set to some N_2 only if the Leader message for N_2 has been sent to it, (2) a Leader message exists only if the sender holds a quorum of votes, (3) a node has recorded a vote only if the corresponding VoteFor message exists, (4) if

a `VoteFor` message exists, then the sender's `hasVoted` flag is set, and (5) two distinct `VoteFor` messages from the same sender cannot exist.

This inductive invariant is surprisingly unwieldy, considering the simple algorithm it describes. The communication state, which was abstract in the implementation, must be detailed and exhaustively related to the node states, which contain essentially the same information. In addition to the effort involved in discovering this invariant, *checking* it is also complicated. We may wish to send the induction verification conditions to an SMT solver, but the domain complexity will make its behavior unpredictable. The global state Σ is essentially a function from node IDs to *sets* of node IDs, since node states contain the `myVotes` field. At the same time, verification relies on a quorum intersection axiom with a $\forall\exists$ quantifier alternation on the same sorts, in the opposite direction:

$$\forall s_1, s_2 \in \texttt{Set(NodeId)}. \quad \texttt{isQuorum}(s_1) \land \texttt{isQuorum}(s_2) \implies (\exists N \in \texttt{NodeId}. \quad N \in s_1 \land N \in s_2).$$

This combination induces a *sort cycle* between `NodeId` and `Set(NodeId)`, disqualifying the verification conditions from the EPR fragment of logic and making their decidability uncertain. The developer must either proceed with fragile automation or expend additional manual effort to abstract the conditions and axioms to get back into EPR [Padon et al. 2017].

2.2 Weakly Replicated State

A node in the election algorithm of Fig. 1a operates using two basic abilities: modification of its `local` state, and sending messages using **send**. This degree of freedom makes it impossible to reason about the system's behavior without revealing and exhaustively constraining its full complexity: the array of node states Σ and the communication history Δ. Intuitively, our novel approach to verified distributed algorithm development breaks this pattern by adopting a higher-level implementation model: *weakly consistent replicated state*.

Fig. 1b details our implementation of a replicated state election algorithm. Rather than separately modifying local state and sending messages, nodes in this system perform *replicated updates* that do both: the statement **update** `Vote`(N_1, N_2) records a vote for N_2 from N_1 in the node's local state, and also broadcasts a message instructing other nodes to make the same change to their states. In contrast to the message-passing algorithm, a node in the replicated-state election algorithm maintains a record of all votes, not just its own votes. The field `local.votes` maps each node that has voted to its candidate. This simplifies the algorithm's logic in two ways. Nodes can check `local.votes` to ensure they have not yet voted, rather than maintaining a separate `hasVoted` field, and each node can check for itself whether a particular candidate has received enough votes to win the election—the leader does not need to send a `Leader` message.

Replicated updates are assumed to be causally consistent: updates can be applied to different nodes in different orders, but if U_1 was visible to the node that created U_2, then U_1 must always be applied before U_2. This assumption provides a crucial balance between efficient implementability and simplification of reasoning. An update is immediately applied to the origin node's state and asynchronously propagated to the node's peers. Unlike the general message-passing model of Fig. 1a, our replicated-state model enforces a distinction in capabilities between *actions* like `castVote` and *handlers*. In the body of an action, the `local` variable is read-only, while in the body of a handler, `local` is read/write but the **update** statement is disallowed.

2.3 Decomposing Consensus Safety

Like the general message passing version, our replicated leader election algorithm is intended to satisfy both the temporal and spatial aspects of consensus safety. While these properties were previously independent, adopting a replicated state model lets us relate them via *convergence*: the often-sought property of weakly consistent replication which requires two nodes that have seen

the same updates to arrive at the same states. Our key observation is that temporal safety, combined with convergence, guarantees spatial safety. Intuitively: if two nodes each recognize a leader, and those nodes cannot un-recognize those leaders, and those nodes must be able to eventually transition to matching states (by delivery of existing updates), then the two leaders must have been the same all along. By decomposing convergence further into *commutativity* between pairs of concurrent updates [Shapiro et al. 2011], we can approach verification of consensus safety—in the weakly replicated state model—using the following rule, where `Monotonicity` refers to just the temporal aspect of safety and `Consensus` refers to full safety:

$$\texttt{Monotonicity} \wedge \texttt{Commutativity} \implies \texttt{Consensus}$$

Crucially, unlike our previous global invariant formalization of consensus, monotonicity and commutativity are conditions on *local* objects. To check monotonicity, we compare each node state to each update. To check commutativity, we consider all update *pairs* in each node state. Thus, we are able characterize consensus safety without referring to *arrays* of node states or messages.

2.4 Verification via Stable Update Preconditions

The remaining challenge is to demonstrate the monotonicity and commutativity properties of replicated leader election (Fig. 1b) without resorting back to the discovery of a complex global inductive invariant. Monotonicity requires an election node to never *undo* its decision about the leader: $\texttt{isLeader}(N_c)$, once true, must remain true in consecutive states. Since state changes are only made by updates, proving monotonicity amounts to showing that the **handle** procedure preserves the existing decision about the leader (if any). Consider a hypothetical execution involving three nodes, $N_{\{1,2,3\}}$ (and quorum-size 2), where **handle** violates monotonicity. The execution starts with the state S_1 with the following votes: $[N_1 \mapsto N_1,\ N_2 \mapsto N_1,\ N_3 \mapsto N_2]$. In this state, $\texttt{isLeader}(N_1)$ already holds. An update $\texttt{Vote}(\texttt{N}_2, \texttt{N}_2)$ now arrives and **handle** is invoked in response. This replaces the vote $N_2 \mapsto N_1$ with $N_2 \mapsto N_2$, thus undoing the earlier decision about the leader. Fortunately, the protocol logic in Fig. 1b preempts this execution by prohibiting double voting. Put differently, we can say that the violating update was not actually *enabled*: $\texttt{Vote}(N_1, N_2)$ can never actually be created-by *or* delivered-to a node that holds state S_1. Given a state S, monotonicity only needs to be checked for updates that are *enabled* in S. Verification therefore requires an artifact that characterizes an enabling condition for updates.

A need for a similar artifact can be felt when verifying commutativity of concurrent updates. In a hypothetical scenario when two updates, $\texttt{Vote}(\texttt{N}_1, \texttt{N}_2)$ and $\texttt{Vote}(\texttt{N}_1, \texttt{N}_3)$, are concurrent, applying them in different orders result in different states regardless of the initial state. Our algorithm also prohibits this scenario: only N_1 can cast a vote in its own name, and all updates that originate from the same node are delivered in their order of creation. The aforementioned updates can therefore never be concurrent and we need an artifact that characterizes the concurrency condition for pairs of updates.

We observe that there is a single artifact that addresses both the aforementioned needs. We call the artifact a *stable update precondition* (SUP), denoted Φ^{s}, which is a relation on node ID, update, and state. Intuitively, $\langle N, U, S\rangle \in \Phi^{\mathsf{s}}(N, E, S)$ means that an update U originating at a node N is enabled in a (local) state S. For our election algorithm, the strong update precondition is as follows:

$$\langle N, \texttt{Vote}(N_1, N_2), S\rangle \in \Phi^{\mathsf{s}} \quad \triangleq \quad N = N_1 \ \wedge \ S[N] = \bot$$

Specifically, (1) the *voter* N_1 indicated by the update must be the same node N that initiated the update, and (2) the state S may not already contain a record of node N voting. Our proof methodology requires the programmer to provide Φ^{s}, analogous to how an inductive proof requires the programmer to prove an inductive invariant. However, considering the high-level abstraction utilized by

our proof technique, SUPs are often simpler than global inductive invariants, as demonstrated by the current example. Sec. 4 elaborates on this difference.

Given Φ^s, our proof technique generates four verification conditions: the first two establish the soundness of Φ^s and the last two verify monotonicity and commutativity using Φ^s. The first condition requires that any state S on node N where `castVote` can generate $\texttt{Vote}(N_1, N_2)$ must satisfy $\langle N, \texttt{Vote}(\texttt{N}_1, \texttt{N}_2), S\rangle \in \Phi^s$. In other words, the enabling condition must be initially true in the state the update is generated. The second condition checks that in any state S where two `Vote` updates are simultaneously enabled, i.e., they are Φ^s-concurrent, they preserve each other's preconditions. This ensures that the enabling condition remains true when the update is applied at a remote replica—notwithstanding any concurrent updates that get there first.

Next, to verify monotonicity, we check that in any state S, if applying an enabled update results in a state S', then the pair $\langle S, S'\rangle$ satisfies the monotonicity condition. This condition is valid for the current example as $\texttt{Vote}(N_1, N_2)$ is enabled in state S only if $S[N_1] = \bot$, i.e., N_1 hasn't yet voted and voting now does not weaken any existing quorum support. Finally, we check that every pair of Φ^s-concurrent messages commute. In the current example, if $\texttt{Vote}(\texttt{N}_1, \texttt{N}_2)$ and $\texttt{Vote}(\texttt{N}_3, \texttt{N}_4)$ are concurrently enabled, then $N_1 \neq N_3$, hence their handlers trivially commute. Leader election therefore satisfies monotonicity and commutativity. Our meta-theory (Secs. 3.1 and 4) now lets us conclude that Leader Election is safe, i.e., there will only ever be a single leader.

3 Consensus from Monotonicity and Commutativity

In this section, we first formalize consensus safety as a property on executions of an abstract distributed system. We then show that, if our system operates using weakly-replicated state updates, we can verify this complex *global* property using conditions on simpler *local* objects: consensus safety can be guaranteed by checking *monotonicity* and *commutativity* for updates to individual node states. We formalize this decomposition in a domain-agnostic way so as to be useful across a wide range of consensus use-cases.

3.1 General Consensus Safety

Different consensus algorithms work with different kinds of decisions. A decision might be a singular, permanent choice, as in the Single-Decree Paxos algorithm [Lamport 1998], or an unbounded stream of values that evolves over the course of the execution, as in the Raft algorithm [Ongaro and Ousterhout 2014]. The common safety requirement for these algorithms is that two *conflicting* decisions never arise in the course of a single execution. We will reason about consensus safety using a general notion of conflict, formalized by the following definition.

Definition 3.1 (Decision Domain). A *decision domain* `Dec` is a set of values, called *decisions*, equipped with a *conflict relation* $(\bowtie) \in \mathcal{P}(\texttt{Dec} \times \texttt{Dec})$ which is symmetric and anti-reflexive. We say that two decisions D_1 and D_2 *agree* when they do not conflict—we write this as $D_1 \bowtie D_2$.

Example 3.2 (Leader Election Decision Domain). Recall the leader election algorithm from Sec. 2. Given a set `Nid` of participating node ids, the decision domain ($\texttt{Dec}_{\texttt{LE}}$) and the conflict relation ($\bowtie_{\texttt{LE}}$) that express the algorithm's consensus semantics are as following:

$$\texttt{Dec}_{\texttt{LE}} \triangleq \{\, \texttt{Undecided} \,\} \cup \{\, \texttt{ElectedLeader}(N) \mid N \in \texttt{Nid} \,\}\,.$$

$$\begin{array}{llll} \texttt{ElectedLeader}(N_1) & \bowtie_{\texttt{LE}} & \texttt{ElectedLeader}(N_2) & \triangleq \quad N_1 \neq N_2 \\ \texttt{Undecided} & \bowtie_{\texttt{LE}} & D & \triangleq \quad \texttt{False} \end{array}$$

To express that only one leader should be elected, we define two decisions containing distinct leader IDs to be in conflict. On the other hand, the initial `Undecided` decision does not conflict with `ElectedLeader`(N): it is safe for nodes to be less up-to-date than their peers. The decision domain for Single-Decree Paxos [Lamport 1998] can be defined similarly to $\texttt{Dec}_{\texttt{LE}}$, using arbitrary values in place of leader IDs.

Example 3.3 (Raft Decision Domain). Unlike leader election and Single-Decree Paxos, Raft [Ongaro and Ousterhout 2014] decides a potentially infinite sequence of values. Its decision domain ($\texttt{Dec}_{\texttt{Raft}}$) and conflict relation ($\bowtie_{\texttt{Raft}}$) can be defined thus:

$$\texttt{Dec}_{\texttt{Raft}} \quad \triangleq \quad \{\, \texttt{Committed}(L) \mid L \in \texttt{List[Entry]} \,\}\,.$$

$$\texttt{Committed}(L_1) \bowtie_{\texttt{Raft}} \texttt{Committed}(L_2) \quad \triangleq \quad \neg(L_1 \textit{ is prefix of } L_2 \textit{ or } L_2 \textit{ is prefix of } L_1).$$

Here, the decision `Committed`$([A, B, C])$ means that entries A, B, and C have been committed to the log, in that order. More up-to-date versions of the log, e.g., $[A, B, C, D]$, and older versions, e.g., $[A, B]$, do not conflict with the current log. However, a log that skips an entry, e.g., $[A, B, D]$, or commits the entries in a different order, e.g., $[A, C, B]$, are considered conflicting. Like Raft, the Multiple-Decree variant of Paxos decides a stream of values, but it allows log entries to be committed out of order, and so the decision domain for Multi-Paxos is a mapping from log indexes to optional entries, where decisions conflict when they have non-equal entries at the same index.

Given a decision domain, we will now formalize *consensus safety* for any distributed system in which nodes adopt decisions from that domain. First, we formally distinguish this class of distributed systems as *decision systems*, and then define them to satisfy consensus safety when each of their executions is free of conflicting decision pairs.

Definition 3.4 (Decision System). Given a decision domain `Dec`, we define a decision system abstractly as a state transition system `DS` that, in every state, assigns a decision $D \in \texttt{Dec}$ to every node $N \in \texttt{Nid}$ in the system. Formally, `DS` is the tuple:

$$\texttt{DS} = \left\langle \texttt{GState}, \texttt{GInit}, \rightsquigarrow, \texttt{Nid}, f^{\mathsf{d}} \right\rangle,$$

where the transition relation $\rightsquigarrow$ relates system states `GState`, starting from initial states $\texttt{GInit} \subseteq \texttt{GState}$. System states represent the global states of a distributed network. The function f^{d} maps network nodes to their decisions: the value $f^{\mathsf{d}}(N, G) \in \texttt{Dec}$ is the decision held by node $N \in \texttt{Nid}$ in global network state $G \in \texttt{GState}$. An *execution* of `DS` is a non-empty sequence of system states, such that the first state is a member of `GInit` and each state G_1 is related to the next state G_2 by $\rightsquigarrow$. For system state G, we define the set $\texttt{ReachFrom}(G) \subseteq \texttt{GState}$ as the set of states reachable from G in zero or more $\rightsquigarrow$ steps. We define the set $\texttt{Reachable}(\texttt{DS}) \subseteq \texttt{GState}$ as the set of states reachable from some initial state in zero or more $\rightsquigarrow$ steps.

Example 3.5. The message-passing leader election algorithm from Sec. 2 is a decision system for $\texttt{Dec}_{\texttt{LE}}$. Each system state G includes an array of nodes, each maintaining a set of votes received from peers and an optional *leader* field. The function $f^{\mathsf{d}}(N, G)$ gives `Undecided` when N's leader field is empty, and gives `ElectedLeader`(N_1) when it is occupied by N_1. The network state G also must include the state of an asynchronous message-passing system, so that one node can send a message into the system in one step and the message can be delivered to another in a later step.

Definition 3.6 (Consensus Safety). A decision system `DS` has the *consensus safety* property iff no pair of conflicting decisions are ever held within a single execution of `DS`:

$$\forall N_1, N_2 \in \texttt{Nid}. \quad \forall G_1 \in \texttt{Reachable}(\texttt{DS}). \quad \forall G_2 \in \texttt{ReachFrom}(G_1). \quad f^{\mathsf{d}}(N_1, G_1) \not\bowtie f^{\mathsf{d}}(N_2, G_2).$$

Note that this definition includes the cases where $N_1 = N_2$ and where $G_1 = G_2$.

Example 3.7 (Consensus Safety for Raft). For our Raft decision domain, consensus safety mandates that if one node ever holds committed log L_1, and another node holds log L_2, then L_1 must be a prefix of L_2 or vice versa. This matches Raft's stated *state machine safety* property, which requires that if one node sees one command committed as the nth entry in the log, then no other node can see a different nth entry [Ongaro and Ousterhout 2014].

Consensus safety is two-dimensional, relating decisions across both the time and space of a distributed execution. A decision held by one node must agree with the decisions held in the past and the future, by both itself and by its peers. As we demonstrated in Sec. 2, this makes verifying consensus safety for message-passing algorithms a difficult task. Even the simple leader election algorithm's global system states are complex objects (see Example 3.5), and consensus safety is a property of *executions*—sequences of those states. The remainder of this section will show how, for decision systems that follow the *replicated update* model, we can derive consensus safety from conditions on local node-state transitions rather than global system transitions.

3.2 Replicated Updates

To achieve local-termed conditions for consensus safety, we now narrow our focus to the class of decision systems that operate in the *replicated state* model, in which every change to one node's state is eventually *replicated* by a similar change to the state of each other node. Replicated state gives us a key advantage: it allows us to specify system behavior in purely local terms, as a transition relation over individual node states.

In this subsection, we will first formalize our particular model of replicated state—*replicated update systems*—and will then formalize our local-termed specifications for these systems.

Definition 3.8. A *replicated update system* is a distributed system in which every node maintains an *application state* and only modifies that state according to *updates* that are communicated to all peers. Each update is initiated by one *origin node*, and carries an *effect* that modifies the application state of each node that receives the update. We define a replicated update system $\mathtt{RS}$ with the tuple:

$$\mathtt{RS} = \left\langle \mathtt{AState}, \mathtt{AInit}, \mathtt{Nid}, \mathtt{Uid}, \mathtt{Action}, \mathtt{Val}, \mathtt{Effect}, f^{\mathsf{h}} \right\rangle,$$

in which $S \in \mathtt{AState}$ is an application state and $\mathtt{AInit} \subseteq \mathtt{AState}$ are the *initial* application states.

$\mathtt{RS}$ defines a transition system in which system states take the form $\langle \Sigma, \Delta \rangle$, with *node configuration* Σ and *update configuration* Δ. The node configuration maps each node ID to its application state and the totally-ordered history of update IDs that it has seen:

$$\Sigma \in \mathtt{Nid} \rightharpoonup (\mathtt{AState} \times \mathcal{P}(\mathtt{Uid})_{\prec}).$$

Given $\Sigma(N) = \langle S, K \rangle$, we use $U_1 \prec_K U_2$ to mean that U_1 precedes U_2 in K's ordering—representing that N applied U_1 before it applied U_2. The update configuration maps each update ID to the update's *origin node*, *effect*, and *causal dependencies*, or to $\bot$ if the update ID has not yet been used:

$$\Delta \in \mathtt{Uid} \rightharpoonup (\mathtt{Nid} \times \mathtt{Effect} \times \mathcal{P}(\mathtt{Uid}))_{\bot}.$$

In an initial transition system state $\langle \Sigma_0, \Delta_0 \rangle$, every node holds an identical initial application state $S_0 \in \mathtt{AInit}$ and has seen no update IDs, and Δ_0 maps every update ID to $\bot$. We use $\mathbf{Init}(S)$ to mean the initial system state defined by initial application state S.

The transitions of the $\mathtt{RS}$ system take two forms, presented in Fig. 2. In the $\textsc{Initiate}(U, N, E)$ transition, node N triggers an update by executing an *action* $f^{\mathsf{a}} \in \mathtt{Action}$. An action represents a request from the node's client or runtime environment. The function $f^{\mathsf{a}} \in (\mathtt{Nid} \times \mathtt{AState} \times \mathtt{Val}) \rightharpoonup (\mathtt{Val} \times \mathtt{Effect}_{\bot})$ takes the local node's ID and application state, as well as any relevant *request value*, and produces a *response value* and optionally an *effect*. For our purposes, we treat the request

$$\textsc{Initiate}(U, N, E)$$
$$\frac{f^{\mathsf{a}} \in \texttt{Action} \qquad f^{\mathsf{a}}(N, S, V_1) = \langle V_2, E\rangle \qquad \Delta(U) = \bot}{\Sigma[N \mapsto \langle S, K\rangle], \Delta \rightsquigarrow \Sigma[N \mapsto \left\langle f^{\mathsf{h}}(E, S), K :: U\right\rangle], \Delta[U \mapsto \langle N, E, K\rangle]}$$

$$\textsc{Deliver}(U, N_u, E, N_d)$$
$$\frac{\Delta(U) = \langle N_u, E, K_u\rangle \qquad U \notin K \qquad K_u \subseteq K}{\Sigma[N_d \mapsto \langle S, K\rangle], \Delta \rightsquigarrow \Sigma[N_d \mapsto \left\langle f^{\mathsf{h}}(E, S), K :: U\right\rangle], \Delta}$$

Fig. 2. Transition rules for a replicated update system. For update ID U and totally-ordered set K, we write $K :: U$ to mean $K \cup \{U\}$ such that U is ordered to the right of all preexisting K elements.

value that comes from the node's client/environment as non-deterministic. Actions that return no effect are invisible to the transition system—the $\textsc{Initiate}(U, N, E)$ transition is only triggered when an action produces an effect.

$\textsc{Initiate}(U, N, E)$ makes two changes to the system state. First, N modifies its application state S by applying E using the handler function $f^{\mathsf{h}} \in (\texttt{Effect} \times \texttt{AState}) \rightharpoonup \texttt{AState}$, and second, it broadcasts this update to its peers. The broadcast is modeled in the transition system by assigning U to $\langle N, E, K\rangle$ in Δ, where U is a fresh effect identifier and K is its causal dependencies—the updates that N saw before creating U.

The second transition is $\textsc{Deliver}(U, N, E)$, which models the delivery of update U to the node N. Updates are delivered to each node exactly once: U can only be delivered if it is not yet in N's update set K. Moreover, updates are delivered in causal order: U can be delivered only if all of its dependencies (K_u) have already been delivered, i.e., $K_u \subseteq K$. This makes RS *causally consistent*—a feature that we will leverage in Sec. 3.4

Example 3.9. The replicated version of the leader election algorithm that we presented in Sec. 2 is an example of a replicated update system, which we can formalize as $\texttt{RS}_{RLE}$. Each state $\langle \texttt{roll}, \texttt{votes}\rangle \in \texttt{AState}_{RLE}$ defines the roll of eligible voters and the record of existing votes: a mapping of voters to their votees. An initial state $S_0 \in \texttt{AInit}_{RLE}$ has an empty vote record: $S_0.\texttt{votes} = \epsilon$. An update's effect adds a single vote to the mapping:

$$\begin{aligned} &\texttt{Effect}_{RLE} &&\triangleq && \{\ \texttt{Vote}(N_1, N_2) \mid N_1, N_2 \in \texttt{Nid}_{RLE}\ \} \\ &{f^{\mathsf{h}}}_{RLE}(\texttt{Vote}(N_1, N_2), S) &&\triangleq && S.\texttt{votes}[N_1 \mapsto N_2] \end{aligned}$$

The eligible voter roll is not modified by any effect. Updates are triggered by two actions that nodes can execute:

$$\begin{aligned} &\texttt{startCampaign}(N, S, \langle\rangle) &&\triangleq && \textbf{if } S.\texttt{votes}(N) = \bot \textbf{ then } \langle \langle\rangle, \texttt{Vote}(N, N)\rangle \textbf{ else } \langle \langle\rangle, \bot\rangle \\ &\texttt{tryGiveVote}(N, S, \langle\rangle) &&\triangleq && \textbf{if } \exists N_1.\ S.\texttt{votes}[N_1] = N_1 \wedge S.\texttt{votes}(N) = \bot \\ &&&&& \textbf{then } \langle \langle\rangle, \texttt{Vote}(N, N_1)\rangle \textbf{ else } \langle \langle\rangle, \bot\rangle \end{aligned}$$

Nodes execute `startCampaign` if they have not seen any votes after a random timeout. Each time a node's state changes, it executes `tryGiveVote` to vote for any other node that has already started a campaign by self-voting. Both actions only return an effect—and thus only initiate a replicated update—if the executing node (N) has not yet voted ($S.\texttt{votes}(N) = \bot$).

Definition 3.10 (Replicated Decision System). If the application states of RS are a decision domain—that is, if we define a conflict relation $\bowtie$ on the $\texttt{AState}_{\texttt{RS}}$ states—then RS is also a decision system (Def. 3.4), as follows. The system states $\langle \Sigma, \Delta\rangle$ of RS define the global state set GState, and likewise

for initial RS system states and the set GInit of initial global states. The transition relation $\rightsquigarrow$ is defined by Fig. 2, and the node decision function f^{d} returns the given node's application state:

$$f^{\mathsf{d}}(N, \langle \Sigma, \Delta \rangle) \quad \triangleq \quad S \quad \text{where} \quad \Sigma(N) = \langle S, K \rangle \,.$$

By this definition, *consensus safety* (Def. 3.6) for RS means that no two conflicting *application states* can arise in any execution of RS.

Example 3.11. As demonstrated in Sec. 2, the replicated leader election algorithm is an example of a replicated decision system. The conflict relation on application states is defined as follows:

$$S_1 \bowtie_{RLE} S_2 \quad \triangleq \quad \exists N_1, N_2. \quad N_1 \neq N_2 \;\wedge\; \texttt{IsLeader}(N_1, S_1) \;\wedge\; \texttt{IsLeader}(N_2, S_2),$$

where $\texttt{IsLeader}(N, S) \triangleq |\{\, N_1 \mid S.\texttt{votes}[N_1] = N \,\}| * 2 \;>\; |\texttt{Nid}|$.

Like our other examples of distributed systems, replicated update systems have complex states: an array of node states, a communication subsystem, etc. However, the replicated model allows us to specify the behavior of a particular system in *local* terms.

Definition 3.12. An *update specification* for a replicated update system RS is a relation $T^{\mathsf{u}} \in \mathcal{P}(\texttt{Nid} \times \texttt{Effect} \times \texttt{AState} \times \texttt{AState})$ such that:

(1) The transition $\textsc{Initiate}(U, N, E)$ over states $\Sigma_1, \Delta_1 \rightsquigarrow \Sigma_2, \Delta_2$, where $\Sigma_1[N] = \langle S_1, K_1 \rangle$ and $\Sigma_2[N] = \langle S_2, K_2 \rangle$, is reachable only if $\langle N, E, S_1, S_2 \rangle \in T^{\mathsf{u}}$.
(2) The transition $\textsc{Deliver}(U, N_u, E, N_d)$ over states $\Sigma_1, \Delta_1 \rightsquigarrow \Sigma_2, \Delta_2$, where $\Sigma_1[N_d] = \langle S_1, K_1 \rangle$ and $\Sigma_2[N_d] = \langle S_2, K_2 \rangle$, is reachable only if $\langle N_u, E, S_1, S_2 \rangle \in T^{\mathsf{u}}$.
(3) T^{u} is *functional*: $\langle N, E, S, S' \rangle \in T^{\mathsf{u}} \wedge \langle N, E, S, S'' \rangle \in T^{\mathsf{u}} \implies S' = S''$.

We say that an update $\langle N, E \rangle$ is *enabled* in S when there exists an S' such that $\langle N, E, S, S' \rangle \in T^{\mathsf{u}}$. When it is clear which T^{u} we are using, we will use $S \xrightarrow{N,E} S'$ to mean that $\langle N, E, S, S' \rangle \in T^{\mathsf{u}}$. Likewise, we will use $S \xrightarrow{N_1,E_1;\ldots;N_n,E_n} S'$ to mean that a sequence of intermediate states exist which the sequence of updates transitions over to reach S':

$$\exists S_1, \ldots, S_{n-1} \in \texttt{AState}. \quad \langle N_1, E_1, S, S_1 \rangle \in T^{\mathsf{u}} \wedge \cdots \wedge \langle N_n, E_n, S_{n-1}, S' \rangle \in T^{\mathsf{u}}.$$

As an example, an update specification for our replicated leader election algorithm could say that no update can modify an existing vote:

$$\langle N, E, S_1, S_2 \rangle \in T^{\mathsf{u}}_{\mathsf{RLE}} \quad \triangleq \quad \forall N_v, N_c \in \texttt{Nid}. \quad S_1.\texttt{votes}(N_v) = N_c \implies S_2.\texttt{votes}(N_v) = N_c.$$

This property is not *local*—we require the context of the overall execution to determine that a $\textsc{Deliver}(U, N_u, E, N_d)$ transition will not violate it. And yet, the update specification allows us to state the property in *local terms*—node IDs, effects, and application states—rather than define it in terms of global system states or executions.

In Sec. 4, we will discuss the definition and verification of suitable update specifications.

3.3 Monotonicity

To understand our first consensus condition, consider an execution of the Leader Election algorithm, in which a node held the Undecided decision, later adopted the $\texttt{Leader}(N_1)$ decision, but subsequently goes back to Undecided. This execution does not violate agreement in the sense two distinct leaders have not been elected, but it is nonetheless prevented by the algorithm. Likewise, the Raft algorithm does not produce executions where a node uncommits previously committed entries of its log. In general, consensus algorithms prevent participants from ever reducing the *strength* of their decisions—we call this property *decision-monotonicity*.

Definition 3.13 (Decision Strength). For decision domain Dec, we define the *strength relation* $\sqsubseteq \in \mathcal{P}(\mathtt{Dec} \times \mathtt{Dec})$ as follows:

$$D_1 \not\bowtie D_2 \quad \wedge \quad \forall D_3.\ D_1 \bowtie D_3 \implies D_2 \bowtie D_3.$$

Two decisions are related by $D_1 \sqsubseteq D_2$ iff the two agree and D_2 conflicts with every decision that D_1 conflicts with. We define $D_1 \sqsubset D_2$ as $D_1 \sqsubseteq D_2 \wedge D_2 \not\sqsubseteq D_1$.

Lemma 3.14. *The strength relation is a preorder.*

Intuitively, $D_1 \sqsubset D_2$ means that D_2 is stronger than D_1, in the sense that D_2 has more conflicts, and is thus more specific. For example, in the decision domain of Raft, $[A, B] \sqsubset [A, B, C]$ because, first, $[A, B, C]$ shares all of $[A, B]$'s conflicts, such as $[A, X]$, and second, $[A, B, C]$ has additional conflicts, such as $[A, B, Z]$ which $[A, B]$ does not conflict with. In the case of Raft, a form of monotonicity is explicitly included, as "Leader Append-Only", among Raft's stated safety properties [Ongaro and Ousterhout 2014]. Paxos and Multi-Paxos also maintain monotonicity, and it is sometimes included in their explicitly-stated safety properties [Lamport 2004].

Definition 3.15. A decision system DS is *decision-monotonic* iff, in each reachable system state, each node's decision is related by $\sqsubseteq$ to all later decisions of the same node:

$$\forall N \in \mathtt{Nid}_{\mathtt{RS}}.\quad \forall G_1 \in \mathtt{Reachable}(\mathtt{DS}).\quad \forall G_2 \in \mathtt{ReachFrom}(G_1).$$
$$G_1(N) = D_1 \ \wedge\ G_2(N) = D_2 \quad \implies \quad D_1 \sqsubseteq D_2.$$

Recall that when $\mathtt{AState}_{\mathtt{RS}}$ is a decision domain, then the replicated update system RS serves as a decision system, in which application states are decisions. Using an update specification for RS, we can derive the decision-monotonicity property from a condition on local terms.

Theorem 3.16. *If* RS *satisfies* T^{u}*, then* RS *is decision-monotonic under the following condition:*

$$\forall \langle N, E, S_1, S_2 \rangle \in T^{\mathsf{u}}.\quad S_1 \sqsubseteq S_2.$$

We refer to any T^{u} that satisfies the condition of Thm. 3.16 as a *decision-monotonic*.

Observe that in a decision-monotonic system, two decisions must not conflict if they are both held by a single node within a single execution, because non-conflict is required for them to be related by $\sqsubseteq$. Thus, monotonicity allows us to guarantee the temporal aspect of consensus safety using a condition on individual update transitions. The spatial aspect of consensus still remains, for which we will define a second condition that relates the behavior of an update across nodes.

3.4 Commutativity

Commutativity is a well-known property in the realm of CRDTs [Shapiro et al. 2011]. In general, *universal* commutativity—the property that *any* pair of replicated application updates commute when applied to *any* application state—is sufficient for consensus safety when combined with monotonicity. For illustration, consider the system execution in Fig. 3, in which two nodes have applied different updates to arrive at different application states: S_1 and S_2. Indeed, S_1 and S_2 need not be identical—they are the products of different updates—but consensus safety requires that they at least agree with each other. We make this argument using commutativity.

The dashed lines in the figure illustrate the hypothetical *completion* of each node's history: its state after applying the updates it has missed. If the system is monotonic (Def. 3.15), then S_3 must be stronger than both S_1 and S_2, and in that case we can deduce $S_1 \not\bowtie S_2$ by applying a lemma:

Lemma 3.17. *If* S_1 S_2*, and* S_3 *are decisions for which* $S_1 \sqsubseteq S_3$ *and* $S_2 \sqsubseteq S_3$*, then* $S_1 \not\bowtie S_2$.

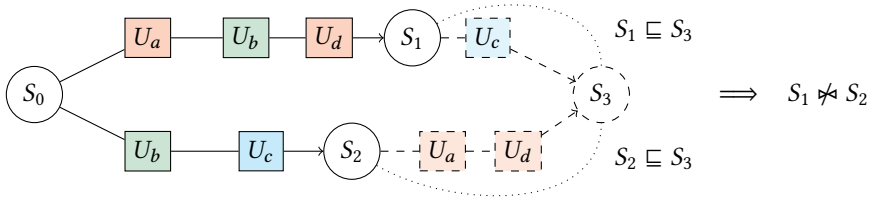


Fig. 3. Monotonicity and commutativity, together, guarantee consensus safety. The solid lines and boxes represent an execution with two nodes, in which U_b was applied by both nodes, but U_a, U_c, and U_d were applied only by their origin nodes. The dashed lines represent a hypothetical extension of the execution, in which the nodes have delivered U_a and U_c to each other. Commutativity ensures that the extensions converge to a common final state: S_3. Monotonicity ensures that S_1 and S_2 are each weaker ($\sqsubseteq$) than S_3, and so S_1 and S_2 cannot be in conflict (Lem. 3.17).

Unfortunately, universal commutativity is too strong for replicated applications implementing consensus. For example, in Replicated Leader Election, the effects $\texttt{Vote}(N_1, N_2)$ and $\texttt{Vote}(N_1, N_3)$ do not commute as they both update $N_1's$ vote in the replicated map of votes. Indeed, these effects *need not* commute as the protocol logic preempts the possibility of their concurrent execution. In case of Leader Election, a node only ever votes in its name and votes by a single node are never concurrent. Likewise in Paxos, there will never be two concurrent proposals in a single term since only one node can get a quorum of acknowledgments required to propose. Universal commutativity is too strong in either case. We define a weaker commutativity constraint that requires a pair of updates to commute only in *racing states* where both are simultaneously *enabled*. Using the update specification T^{u}, we formalize this intuition below.

Definition 3.18 (Racing-State Commutativity). An update specification T^{u} is *racing-state commutative* iff, for every pair of distinct-origin updates that are both enabled for a single application state S, there exists a common state S' that can be reached by applying the updates to S in either order:

$$\forall N_l, N_r \in \texttt{Nid}. \quad \forall E_l, E_r \in \texttt{Effect}. \quad \forall S, S_l, S_r \in \texttt{AState}.$$

$$N_l \neq N_r \quad \wedge \quad S \xrightarrow{N_l, E_l} S_l \quad S \xrightarrow{N_r, E_r} S_r \quad \implies \quad \exists S' \in \texttt{AState}. \quad S_l \xrightarrow{N_r, E_r} S' \quad S_r \xrightarrow{N_l, E_l} S'$$

Intuitively, racing-state commutativity applies to all *concurrent* update pairs that arise in an execution—those that can appear in different orders in different contexts. Because replicated update systems enforce causal consistency, a pair of updates can only appear in different orders when they originate from distinct nodes.

We will now prove that the convergent state S_3 in Fig. 3 exists, while only assuming that the depicted system satisfies an update specification T^{u} which is racing-state commutative. We do so by transforming the concrete execution (the part with solid lines) into the hypothetical execution in Fig. 4. Our first step is to factor out the two nodes' *common history*, so that both nodes start by taking the same step $S_0 \xrightarrow{U_b} S$ for some S. To do so, we apply our racing-state-commutativity assumption: U_a and U_b are racing on state S_0, and so we can flip them in the top node's history without changing their post-state. This step proves that T^{u} includes the dashed $S \longrightarrow S_1$ segment.

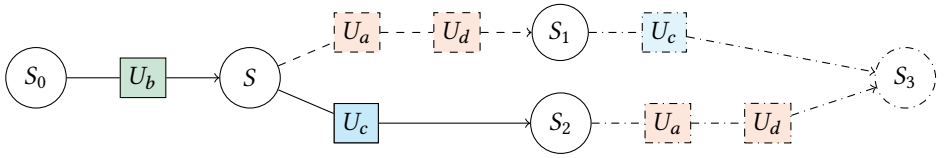


Fig. 4. Transformation of the execution in Fig. 3, using the racing-state commutativity assumption to ensure the existence of a common S_3. Lem. 3.23 ensures that T^{u} includes the dashed segments of the transformed execution, and Lem. 3.19 ensures that T^{u} includes the dash-dotted segments. The segment $S_0 \to S \to S_2$ is solid because it is part of the concrete execution in Fig. 3.

Now, showing that S_3 exists simply requires us to commute U_c with the *sequence* $[U_a; U_d]$. In fact, racing-state commutativity allows us to commute sequences, according to the following lemma.

Lemma 3.19 (Commutative Sequences). *If an update specification T^{u} is racing-state commutative, then two sequences of updates commute when there is no origin node that appears in both sequences and when both sequences are enabled for the same state:*

$$\forall N_1^l, \ldots, N_n^l, N_1^r, \ldots, N_m^r \in \mathtt{Nid}. \quad \forall E_1^l, \ldots, E_n^l, E_1^r, \ldots, E_m^r \in \mathtt{Effect}. \quad \forall S, S_l, S_r \in \mathtt{AState}.$$

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m} (N_i^l \neq N_j^r) \quad \wedge \quad S \xrightarrow{N_1^l, E_1^l; \ldots; N_n^l, E_n^l} S_l \quad S \xrightarrow{N_1^r, E_1^r; \ldots; N_m^r, E_m^r} S_r \quad \Longrightarrow \quad \exists S' \in \mathtt{AState}. \; S_l \xrightarrow{N_1^r, E_1^r; \ldots; N_m^r, E_m^r} S' \quad S_r \xrightarrow{N_1^l, E_1^l; \ldots; N_n^l, E_n^l} S'$$

We know that U_c in Fig. 3 does not share an origin with either U_a or U_d, because otherwise it would appear always-before or always-after those updates. Thus, we can apply Lem. 3.19 to ensure the existence of S_3. And thus, if T^{u} is decision-monotonic, we also ensure that $S_1 \not\bowtie S_2$.

Racing-state commutativity always allows us to perform the common-history factoring step. The following lemmas state this, relying on the fact that replicated update system executions are causally-consistent. When working in the context of a specific system state, we will use the shorthand $S \xrightarrow{K} S'$ to mean that $S \xrightarrow{N_1, E_1; \ldots; N_n, E_n} S'$ for the origin nodes and effects of the updates in K, as defined by the system state.

Definition 3.20 (Causal Cut). Given update configuration Δ, we say that history K is a *causal cut* of Δ iff for every $U \in K$, there is an entry $\Delta(U) = \langle N, E, K_1 \rangle$ such that $K_1 \subseteq K$ and $\forall U_1 \in K_1.\ U_1 <_{K_1} U$.

Lemma 3.21 (Causal Consistency). *For any* RS*, every history contained in every reachable system state is a causal cut of that state's update configuration.*

Lemma 3.22 (Causal Cut Enabled). *Given system* RS*, which satisfies T^{u} and has system state $\langle \Sigma, \Delta \rangle$ reachable from* **Init**(S_0)*, if K is a causal cut of Δ, then $\exists S.\ S_0 \xrightarrow{K} S$.*

Lemma 3.23 (Common History). *For system state $\langle \Sigma, \Delta \rangle$ reachable from S_0, in which $\Sigma(N_a) = \langle S_a, K_a \rangle$ and $\Sigma(N_b) = \langle S_b, K_b \rangle$, let $H = K_a \cap K_b$, such that H has the same ordering as K_a. If T^{u} is racing-state commutative, then $S \in \mathtt{AState}$ exists for which $S_0 \xrightarrow{H} S$, $S \xrightarrow{K_a \setminus K_b} S_a$, and $S \xrightarrow{K_b \setminus K_a} S_b$.*

Intuitively, once we have factored out the common history of two nodes using Lem. 3.23, we can commute their remaining non-common suffixes to reach a common final state, like S_3 in Fig. 3.

Monotonicity ensures that the common final state is a $\sqsubseteq$-upper-bound for the nodes' existing states, and so those states must agree.

Theorem 3.24 (Consensus from Monotonicity and Commutativity). *If T^{u} is decision-monotonic and racing-state commutative, then every T^{u}-system* RS *has the consensus safety property.*

4 Verification via Stable Update Preconditions

In this section, we consider the formulation of sound update specifications for given systems.

4.1 Stable Update Precondition

Def. 3.12 requires $T^{\mathsf{u}}(N, E, \cdot, \cdot)$ to relate two distinct pairs of states capturing the effect of applying E from N at origin and remote nodes respectively. In both instances, the final state S_2 is obtained from the initial state S_1 by applying the handler, $S_2 = f^{\mathsf{h}}(E, S_1)$. Thus $T^{\mathsf{u}}(N, E, S_1, S_2)$ could simply be defined as a relational abstraction of $f^{\mathsf{h}}(E, \cdot)$. However, simply capturing the semantics of the handler function makes T^{u} too weak to be useful: RS does not admit all transitions defined by f^{h}. In particular, the Initiate transition (Fig. 2) can only apply f^{h} to a state S_1 if the system includes an *action* f^{a} that returns an E when given S_1. From this, we can derive a notion of *local precondition* $\Phi(N, E)$ for an update with origin N and effect E: a condition that S_1 must satisfy for any action $f^{\mathsf{a}}(N, S_1, V)$ to produce E, and thus initiate the update $\langle N, E\rangle$. For example, in replicated leader election, N can produce $\mathtt{Vote}(N_1, N_2)$ in a state S_1 only if $N = N_1$ and if N has not already voted in S_1. We could then define $\langle N, E, S_1, S_2\rangle \in T^{\mathsf{u}}$ iff $S_1 \in \Phi(N, E)$ and $f^{\mathsf{h}}(E, S_1) = S_2$.

The T^{u} based on local precondition is correct and useful in the case of replicated leader election, but for many systems, it is actually *under*-approximate, and thus unsafe: it excludes transitions that the system may actually take. The *local* precondition is only guaranteed to hold in the Initiate step—in the following Deliver steps, E may be applied to different states where other concurrent updates have already taken effect. For example, consider a replicated Paxos-like system, in which node N campaigns to be leader of round r and then proposes the decision value v using an update $\mathtt{Propose}(r, v)$. Before generating $\mathtt{Propose}(r, v)$, N checked that its state did not contain a preexisting proposal, and so $S \in \Phi(N, \mathtt{Propose}(r, v))$ could require $S.\mathtt{proposal} = \bot$. However, when the update is delivered to node N_d, it is possible that N_d has already seen an alternative proposal, and $S_d.\mathtt{proposal} = \langle r_1, v_1\rangle$. For this system, Φ defines an unsafe T^{u}—to define a safe, over-approximate T^{u}, we must discover a weaker, *stable* precondition, which holds for the pre-state of each Initiate step *and* each Deliver step. For example:

$$S \in \Phi^{\mathsf{s}}(N, \mathtt{Propose}(r, v)) \quad \triangleq \quad S.\mathtt{proposal} = \langle r_1, v_1\rangle \implies r_1 \neq r$$

A *stable update precondition* (SUP) over-approximates the condition under which an effect can be applied to a state, during either an Initiate transition or a Deliver transition.

Definition 4.1 (Stable Update Precondition). A replicated update system RS satisfies the *stable update precondition* $\Phi^{\mathsf{s}} \in \mathcal{P}(\mathtt{Nid} \times \mathtt{Effect} \times \mathtt{AState})$ iff $\langle N, E, S\rangle \in \Phi^{\mathsf{s}}$ whenever an update with effect E and origin node N modifies state S at any node. Specifically, RS satisfies the following:

(1) For any reachable $\textsc{Initiate}(U, N, E)$, node N starts with S such that $\langle N, E, S\rangle \in \Phi^{\mathsf{s}}$.
(2) For any reachable $\textsc{Deliver}(U, N_u, E, N_d)$, node N_d starts with S such that $\langle N_u, E, S\rangle \in \Phi^{\mathsf{s}}$.

For readability, we will also write $S \in \Phi^{\mathsf{s}}(N, E)$ to mean $\langle N, E, S\rangle \in \Phi^{\mathsf{s}}$.

Using a stable update precondition for RS, we can now define a non-trivial, over-approximate update specification for RS:

$$\langle N, E, S_1, S_2\rangle \in T^{\mathsf{u}}(\mathsf{RS}, \Phi^{\mathsf{s}}) \quad \triangleq \quad S_1 \in \Phi^{\mathsf{s}}(N, E) \ \wedge \ S_2 = f^{\mathsf{h}}_{\mathsf{RS}}(E, S_1)$$

Theorem 4.2. *For replicated update system* RS *with stable update precondition* Φ^{s}*, the relation* $T^{\mathsf{u}}(\mathsf{RS}, \Phi^{\mathsf{s}})$ *is an update specification for* RS.

By applying our monotonicity (Def. 3.15) and commutativity (Def. 3.18) conditions to $T^{\mathsf{u}}(\mathsf{RS}, \Phi^{\mathsf{s}})$, we realize Φ^{s} as the key artifact required to verify consensus safety for RS. As the example demonstrates, a suitable SUP does not follow directly from the system's local behavior—it must be creatively discovered. Suitable SUPs must be creatively discovered, and are thus analogous to a global inductive invariant in standard verification approaches, but with a key difference: a global invariant is defined in *global* terms—its domain is system states, structures which contain *many* node states and *many* messages. In contrast, an SUP is defined in *local* terms—it relates an *individual* node ID and effect to an *individual* node state. Let `AState` denote the type of a node state in any (distributed) decision system.

4.2 Verification Conditions for SUPs

The following theorem formalizes local-termed conditions for verifying an SUP (and thus the update specification based upon it).

Theorem 4.3 (SUP Verification). *A replicated update system* RS *has stable update precondition* Φ^{s} *if the following verification conditions hold:*

Initiation Safety

$$\forall N \in \mathtt{Nid}_{\mathsf{RS}}.\quad \forall E \in \mathtt{Effect}_{\mathsf{RS}}.\quad \forall S \in \mathtt{AState}_{\mathsf{RS}}.\quad \forall V_1, V_2 \in \mathtt{Val}_{\mathsf{RS}}.\quad \forall f^{\mathsf{a}} \in \mathtt{Action}.$$
$$f^{\mathsf{a}}(N, V_1, S) = \langle V_2, E\rangle \implies \langle N, E, S\rangle \in \Phi^{\mathsf{s}}$$

Racing-State Stability

$$\forall N_1, N_2 \in \mathtt{Nid}_{\mathsf{RS}}.\quad \forall E_1, E_2 \in \mathtt{Effect}_{\mathsf{RS}}.\quad \forall S, S' \in \mathtt{AState}_{\mathsf{RS}}.$$
$$\langle N_1, E_1, S\rangle \in \Phi^{\mathsf{s}} \wedge \langle N_2, E_2, S\rangle \in \Phi^{\mathsf{s}} \wedge f^{\mathsf{h}}_{\mathsf{RS}}(E_2, S) = S' \implies \langle N_1, E_1, S'\rangle \in \Phi^{\mathsf{s}}$$

Racing-State Commutativity

$$\forall N_1, N_2 \in \mathtt{Nid}_{\mathsf{RS}}.\quad \forall E_1, E_2 \in \mathtt{Effect}_{\mathsf{RS}}.\quad \forall S, S_1', S_1'', S_2', S_2'' \in \mathtt{AState}_{\mathsf{RS}}.$$
$$(\langle N_1, E_1, S\rangle \in \Phi^{\mathsf{s}} \wedge \langle N_2, E_2, S\rangle \in \Phi^{\mathsf{s}} \wedge f^{\mathsf{h}}_{\mathsf{RS}}(E_1, S) = S_1'$$
$$\wedge f^{\mathsf{h}}_{\mathsf{RS}}(E_2, S_1') = S_1'' \wedge f^{\mathsf{h}}_{\mathsf{RS}}(E_2, S) = S_2' \wedge f^{\mathsf{h}}_{\mathsf{RS}}(E_1, S_1') = S_2'') \implies S_1'' = S_2''$$

The first condition simply checks that Φ^{s} holds for each update at the moment that it is created. The second condition checks that an update's precondition is not invalidated if another racing update is delivered first. We use the same notion here of *racing states* that we used in Sec. 3.4: a state that is racing for Φ^{s} (when it satisfies Φ^{s} for two distinct-origin updates) is also racing, by our previous definition, for $T^{\mathsf{u}}(\mathsf{RS}, \Phi^{\mathsf{s}})$. The third condition is the SUP equivalent to our *racing-state commutativity* condition on update specifications (Def. 3.18) The ability to commute concurrent updates is not only useful to ensure consensus safety—it is foundational to how we verify the stability of Φ^{s}, and thus the correctness of $T^{\mathsf{u}}(\mathsf{RS}, \Phi^{\mathsf{s}})$.

For a SUP Φ^{s} verified using Thm. 4.3, the resulting $T^{\mathsf{u}}(\mathsf{RS}, \Phi^{\mathsf{s}})$ already satisfies racing-state commutativity. Thus, we achieve consensus safety using one additional condition which ensures that $T^{\mathsf{u}}(\mathsf{RS}, \Phi^{\mathsf{s}})$ satisfies decision-monotonicity (Def.3.15).

Corollary 4.4 (Consensus Safety Verification). *A replicated update decision system* RS*, for which* Φ^{s} *satisfies the conditions of Thm. 4.3, has consensus safety if the following holds:*

Decision Monotonicity

$$\forall N \in \mathtt{Nid}_{\mathsf{RS}}.\ \forall E \in \mathtt{Effect}_{\mathsf{RS}}.\ \forall S_1, S_2 \in \mathtt{AState}_{\mathsf{RS}}.\ S_1 \in \Phi^{\mathsf{s}}(N, E) \wedge f^{\mathsf{h}}_{\mathsf{RS}}(S_1) = S_2 \implies S_1 \sqsubseteq S_2$$

Our approach thus generates four verification conditions—all using purely local terms—to establish the consensus safety of a replicated update system.

4.3 Decidability

To support automated verification, we observe that our verification conditions for consensus safety can be adapted to fit into the decidable *Extended EPR* [Taube et al. 2018] fragment of logic. In this form, the conditions are checked against *relational abstractions* of a system's actions (f^{a}) and handler function (f^{h}).

Definition 4.5. The relations $R^a \in \mathcal{P}(\mathtt{Nid} \times \mathtt{AState} \times \mathtt{Val} \times \mathtt{Effect})$, $R^h \in \mathcal{P}(\mathtt{Effect} \times \mathtt{AState} \times \mathtt{AState})$, and $R^d \in \mathcal{P}(\mathtt{AState} \times \mathtt{AState})$ *abstract* RS under the following conditions:

(1) R^a over-approximates the set of actions:

$$\forall f^{\mathsf{a}} \in \mathtt{Action}_{\mathsf{RS}}.\ \forall N \in \mathtt{Nid}_{\mathsf{RS}}.\ \forall S \in \mathtt{AState}_{\mathsf{RS}}.\ \forall V_1, V_2 \in \mathtt{Val}_{\mathsf{RS}}.$$
$$f^{\mathsf{a}}(N, S, V_1) = \langle V_2, E \rangle \implies \langle N, S, V_1, E \rangle \in R^a.$$

Note that R^a abstracts away the V_2 action output, which is inconsequential to state safety.

(2) R^h over-approximates the handler function:

$$\forall E \in \mathtt{Effect}_{\mathsf{RS}}.\ \forall S \in \mathtt{AState}_{\mathsf{RS}}.\ \forall S' \in \mathtt{AState}_{\mathsf{RS}}.\ f^{\mathsf{h}}(N, S) = S' \implies \langle N, S, S' \rangle \in R^h.$$

(3) R^d *under*-approximates the decision strength relation:

$$\forall S_1, S_2 \in \mathtt{Effect}_{\mathsf{RS}}.\ \langle S_1, S_2 \rangle \in R^d \implies S_1 \sqsubseteq S_2.$$

Extended EPR requires that sorts are *stratified*, i.e. totally ordered such that a function symbol $f \in A \to B$ requires $A > B$, and a quantifier alternation of the form $\exists A.\ \forall B$ also requires $A > B$.

Definition 4.6. We obtain the *decidability-friendly* verification conditions by modifying the four conditions given in Thm. 4.2 and Cor. 4.4:

(1) Replace each instance of $f^{\mathsf{a}}(N, S, V_1) = \langle V_2, E \rangle$ with $\langle N, S, V_1, E \rangle \in R^a$.
(2) Replace each instance of $f^{\mathsf{h}}(E, S_1) = S_2$ with $\langle E, S_1, S_2 \rangle \in R^h$.
(3) Replace each instance of $S_1 \sqsubseteq S_2$ with $\langle S_1, S_2 \rangle \in R^d$.

Theorem 4.7. *If* $\langle R^a, R^h, R^d \rangle$ *abstracts* RS*, and the conditions of Def. 4.6 hold for some* Φ^{s}*, then* RS *has the consensus safety property.*

Of course, the ultimate decidability of the decidability-friendly conditions depends on the inputs—R^a, R^h, R^d, and Φ^{s}—themselves being sort-stratified. The advantage, compared with traditional verification conditions on global system states, is that our conditions do not mandate any stratification orderings. Intuitively, global states map nodes to their states, and so a verification condition over a global state would force the ordering $N > \mathtt{AState}$. If the condition's inputs defined a contradictory ordering $N < \mathtt{AState}$, the decidability of the condition would be in question.

Theorem 4.8 (Decidability Friendly). *The Def. 4.6 conditions are decidable as long as the relations* R^a*,* R^h*,* R^d*, and* Φ^{s}*, taken together, are in the Extended EPR logic fragment.*

In the next section, we discuss our case study, in which we used our tool Super-V to generate these abstractions. Existing tools such as IVy [Taube et al. 2018] are also suitable to this purpose. The details of abstracting sequential code in this way are beyond the scope of this paper.

5 Case Study: Log Replication

To evaluate the practicality of our new approach, we used the weakly-replicated application programming model to implement a log consensus algorithm we call *Ferry*. Ferry's behavior (Sec. 5.1) is extensionally identical to the the well-known Raft algorithm [Ongaro and Ousterhout 2014]. Internally, we leveraged the underlying weak replication model to simplify Ferry's design compared to Raft (Sec. 5.2), and we applied our automated verification technique to verify consensus safety for Ferry using less effort than required for Raft (Sec. 5.3). We compared the runtime performance of Ferry to that of an industry-strength Raft implementation, and found that Ferry's design does not incur any significant performance penalty (Sec. 5.5).

Ferry demonstrates that our replicated application approach provides a significant advantage—a reduction of verification effort—at no significant cost to performance.

5.1 Safety and Fault-Tolerance

Ferry replicates a growing log of values across a network of nodes. The log is guaranteed to be append-only: once one node sees a value committed to the log, no other node will see a different value in its place. To illustrate this property, consider the network of nodes in Fig. 5. If we ignore N_4, then the remaining nodes N_1, N_2, and N_3 satisfy the guarantee: no two nodes contain different values for the same log index. Node N_4, however, disagrees with nodes N_2 and N_3 on the value at index 2.

Append-only logs make Ferry, like Raft, suitable to host a *replicated state machine*, in which nodes each maintain a copy of a state machine and use the log as a stream of commands. Nodes apply the commands to their machines—irrevocably—as the commands are committed to the log. The append-only guarantee, called *state machine safety* in the context of Raft, ensures that each node's state machine follows the same sequence of transitions.

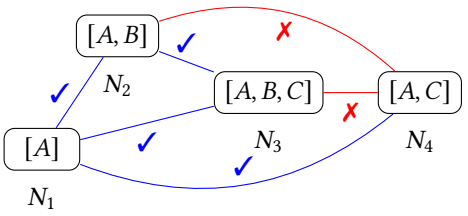


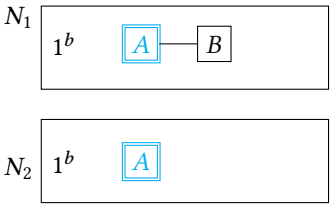
Fig. 5. A network of nodes holding copies of a replicated log. Blue check-marks denote agreement, and red cross-marks denote conflict.

We formalized the append-only property as an instance of general consensus safety (Sec. 3.1), using Ferry's node states as the *decisions* for which consensus safety must be maintained. Our instantiation defines node state S_2 to be *stronger than or equivalent to* state S_1 iff S_1's committed log is a prefix of S_2's committed log. By this definition, two Ferry nodes are *in conflict*—violating safety—when they hold committed logs such that one is not a prefix of the other, which is consistent with our examples of the append-only property in Fig. 5.

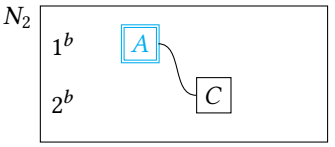
Ferry provides the same key fault-tolerance property as Raft: it is always possible to make progress, i.e., commit entries to the replicated log, as long as a majority of the nodes are functioning and able to communicate. No distributed consensus algorithm can guarantee progress within a finite amount of time or number of steps [Fischer et al. 1985], but Ferry, like Raft, uses random timeouts to make fast progress highly probable. We do not address the formal verification of fault-tolerance or progress in this paper, but we do empirically evaluate the speed with which Ferry makes progress in the presence of failures in Sec. 5.5.

5.2 Design and Implementation

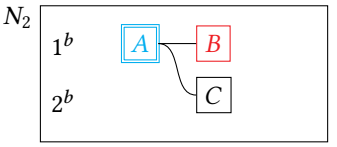
To maintain fault tolerance, Ferry nodes must be able to operate concurrently. However, concurrent additions to the log can violate safety, creating conflicts like the one between N_2 and N_4 in Fig. 5.



(a) State after N_1 proposes A and B. N_2 receives only A, which it accepts.



(b) N_2's state after appending C.



(c) N_2's state after receiving B from N_1.

Fig. 6. Copies of the replicated log at nodes N_1 and N_2. Cyan entries with double border are committed. Others are proposed but not committed.

To avoid these conflicts, Ferry, like Raft, requires nodes to first *propose* each new entry to their peers and commit it to the log only after receiving a quorum of acknowledgments. This ensures that only one entry can be committed—by any number of nodes—at any one index. Each node tracks the progress of the proposals it has seen thus far. Unlike Raft, Ferry uses a tree data structure that leverages weak replicated state to maintain alternative pending proposals side-by-side. We demonstrate this mechanism in Fig. 6 and Fig. 7, using an example execution of a Ferry cluster with three nodes: N_1, N_2, and N_3. Initially, node N_1 adds the entry A to the log, which is accepted by all nodes, hence committed. N_1 then proposes B as the next entry, but a transient network partition prevents other nodes from receiving this proposal. At this point, the nodes N_1 and N_2 have the replicated state copies shown in Fig. 6a. Here, 1^b denotes the branch number, which is analogous to a term number in Raft. Just like Raft has a unique leader for a term, Ferry has a unique owner for a branch. The existing entries (A and B) are both on branch 1^b, which N_1 owns.

Suppose the network partition continues, and N_2, assuming that N_1 has failed, decides to take on the role of proposing new entries. To do so, N_2 must take ownership of a new, higher-numbered branch, by convening an election similar to that described in Sec. 2. Assume N_2 wins this election and becomes the owner of branch 2^b. N_2 then issues the $\texttt{Propose}(1^b, 1^i, 2^b, C)$ update, which initializes the branch 2^b with the entry C. The first two arguments, 1^b and 1^i, denote the branch and index of a previous entry in the log. In this case, C is added so that it follows A—the entry at 0^i index in branch 0^b—which is the latest entry that N_2 had seen. The state of N_2 at this point is shown in Fig. 6b.

Assume that the network partition is repaired and the update $\texttt{Propose}(1^b, 1^i, 1^b, B)$, issued by N_1 to propose entry B, is finally delivered to N_2. Even though B is on the old branch 1^b, N_2 still adds the entry to its tree. This is because C has not yet been majority-accepted, and so B could still become part of the committed log. The state of N_2 at this point is shown in Fig. 6c. The entry B is colored red to show that B has been *deferred*: it is not currently part of N_2's proposed log—it has been *bypassed* by the entry C on a higher branch—but it can still become part of the proposed or committed log in the future if an entry on a branch higher than 2^b cites B as its previous entry.

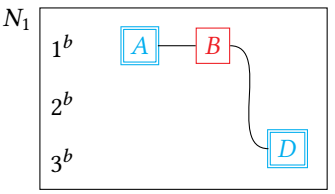


Fig. 7. The state of N_1 after becoming owner of 3^b, proposing D on 3^b, and receiving an Accept for D from N_3.

Assume that a new network partition now separates N_2 from rest of the cluster, preventing its peers from receiving its C proposal, and that N_1 wins an election in the aftermath to establish

a new branch 3^b. N_1 now proposes a new entry D on 3^b to be included in the log following its earlier proposal B, using update `Propose`$(1^b, 2^i, 3^b, D)$. Node N_3 receives the proposal, and accepts it by responding with `Accept`$(3^b, N_3, 3^i)$. Assuming N_1 accepted its own proposal, this gives the proposal D a majority acceptance, leading to D and its prior entry B being committed (causal consistency guarantees that wherever D is delivered, B has also been delivered). The state of N_1 at this point is as shown in Fig. 7. Observe that B, which was deferred, is now part of N_1's committed log $[A, B, D]$: the path from the tree's root to the highest-index entry with majority acceptance. When N_2 eventually receives D's `Propose` and `Accept` updates, it will adopt the same committed log $[A, B, D]$, and D will have bypassed C in N_2's tree. At that point, because C has been bypassed by a *committed* entry, C can no longer become part of any proposed log, and is eligible for garbage collection.

Comparison to Raft. As described earlier, Ferry's design is inspired by Raft and adapts it to the replicated state setting. There are, however, several notable differences.

First, Raft nodes only maintain a linear log of proposals, not a tree. In a Raft execution that follows similarly to the Ferry example we gave above, N_2 would discard the B proposal it received, rather than store it (deferred) as illustrated in Fig. 6c. For this reason, Raft's design requires *retransmission*: for N_2 to receive the final D proposal from N_1, it would need to request that N_1 retransmit the B entry. Ferry leverages its underlying causal replication layer to ensure that updates are, eventually, transmitted once. It then uses the tree structure to save potentially future-relevant entries so that they do not need to be retransmitted.

Second, Ferry uses its causal delivery assumption to ensure that new branch leaders have seen all majority-accepted entries on lower branches: to receive the necessary `Vote` updates from its voters, a new branch owner must have seen those voters' pre-existing `Propose` and `Accept` updates. Raft enforces an equivalent requirement by adding explicit metadata—previous-terms and log indexes—to requests for votes.

Third, Ferry nodes use their own state copies to determine when new entries become committed, based on the `Accept` updates that have been broadcast. In contrast, Raft nodes must wait for a commit notification to be sent directly from the current leader. If the leader has failed, a majority-replicated entry cannot be considered committed until a new leader has been elected.

Finally, Ferry employs the *flexible quorums* design [Howard et al. 2017], allowing configurations that trade faster performance under normal conditions for slower performance under failures—to our knowledge, Ferry is the first verified consensus implementation that has this feature. To compare performance in a more straightforward way, we use a Raft-like simple majority configuration in our Sec. 5.5 experiments.

5.3 Safety Verification

We verified Ferry's consensus safety—and thus its append-only guarantee—by using Super-V to instantiate and discharge the verification condition templates of Sec. 4. Compared with existing projects that have verified Raft, our verification of Ferry required significantly less manual effort for two reasons. First, the manual artifacts that we had to provide were more concise than those that have been required for Raft. Second, our instantiated verification conditions fell immediately into a decidable fragment of logic (Extended EPR)—when automated verification has been used for Raft, decidability has required the creative definition and verification of auxiliary abstractions.

Concise Artifacts. Verifying Ferry required us to manually discover *stable update preconditions* (SUPs) for each of Ferry's updates (`Vote`, `Propose`, and `Accept`). SUPs are analogous in function to the global inductive invariants required for traditional verification of distributed systems: they observe restrictions on a system's overall behavior that do not follow directly from local context.

$$\langle N_u, \texttt{Vote}(b, N_{\texttt{voter}}, N_{\texttt{cand}}), S\rangle \in \Phi^{\texttt{s}} \quad \triangleq \quad N_{\texttt{voter}} = N_u \ \wedge \ S.\texttt{votes}[\langle b, N_u\rangle] = \bot$$

$$\begin{aligned}\langle N_u, (\texttt{Propose}(b_{\texttt{old}}, i_{\texttt{old}}, b_{\texttt{new}}, A)), S\rangle \in \Phi^{\texttt{s}} \quad \triangleq \\ \texttt{elected}(N_u, b_{\texttt{new}}, S) \\ \wedge \ \langle b_{\texttt{old}}, i_{\texttt{old}}\rangle \in S.\texttt{tree} \ \wedge \ \langle b_{\texttt{new}}, i_{\texttt{old}} + 1\rangle \notin S.\texttt{tree} \\ \wedge \ (\forall \langle b, i\rangle \in S.\texttt{tree}. \quad \langle b, i\rangle \sqsubseteq_{S.\texttt{tree}} \langle b_{\texttt{old}}, i_{\texttt{old}}\rangle \ \vee \ \texttt{deferred}(b, i, S))\end{aligned}$$

$$\left\langle N_u, \texttt{Accept}(N_{\texttt{accepter}}, b, i), S\right\rangle \in \Phi^{\texttt{s}} \quad \triangleq \quad N_{\texttt{accepter}} = N_u \ \wedge \ \langle b, i\rangle \in S.\texttt{tree} \ \wedge \ \neg\texttt{frozen}(N_u, b, S)$$

$$\texttt{elected}(N, b, S) \quad \triangleq \quad |\{ N_v \mid N_v \in S.\texttt{voters} \wedge S.\texttt{votes}[\langle b, N_v\rangle] = N \}| \ \geq \ S.\texttt{voteQ}$$

$$\texttt{deferred}(b, i, S) \quad \triangleq \quad |\texttt{rejecters}| \ \geq \ S.\texttt{acceptQ}$$

$$\textit{where} \ \texttt{rejecters} \triangleq \{ N \mid N \in S.\texttt{voters} \ \wedge \ \texttt{frozen}(N, b, S) \ \wedge \ S.\texttt{accepts}[\langle b, N\rangle] < i \}$$

$$\texttt{frozen}(N, b, S) \quad \triangleq \quad \exists b_2. \quad b_2 > b \wedge S.\texttt{votes}[\langle b_2, N\rangle] \neq \bot$$

Fig. 8. Stable update preconditions (SUPs) for Ferry. For node state S, the field $S.\texttt{votes}$ maps a branch b and node N to the candidate that N voted to be owner of b, or to $\bot$ if none. The field $S.\texttt{accepts}$ maps b and N to the index of the latest entry that N has accepted on branch b.

However, SUPs are more concise than global invariants because they only relate individual updates to individual states.

The SUPs for Ferry are detailed in Fig. 8. The $\texttt{Vote}(b, N_{\texttt{voter}}, N_{\texttt{cand}})$ update, which records a vote for the owner of branch b, has the same SUP that was described for the one-shot election algorithm in Sec. 2. The SUP for $\texttt{Accept}(N_{\texttt{accepter}}, b, i)$ is similar, requiring that the origin node is the accepting node, that the accepted tree location is occupied, and that the node has not already voted for a higher branch (which represents a promise to stop accepting on lower branches). The SUP for $\texttt{Propose}(b_{\texttt{old}}, i_{\texttt{old}}, b_{\texttt{new}}, A)$ can be broken down into four conditions. First, the proposing node (N_u) must be the elected owner of $b_{\texttt{new}}$. Second, the preceding tree location $\langle b_{\texttt{old}}, i_{\texttt{old}}\rangle$ must be occupied by an entry, and third, the new tree location $\langle b_{\texttt{new}}, i_{\texttt{old}} + 1\rangle$ must be unoccupied.

The fourth condition for Propose is the least trivial, requiring that existing entries *bypassed* by the new entry are *deferred*, i.e. cannot be committed directly. The condition $\langle b, i\rangle \sqsubseteq_{S.\texttt{tree}} \langle b_{\texttt{old}}, i_{\texttt{old}}\rangle$ means that $\langle b, i\rangle$'s log (the sequence of entries from the root to $\langle b, i\rangle$) is a prefix of $\langle b_{\texttt{old}}, i_{\texttt{old}}\rangle$'s log. If that's not the case, then the new proposed log will be missing some of $\langle b, i\rangle$'s entries (for example, B in Fig. 6c), and it is important that those entries cannot concurrently become committed. The $\texttt{deferred}(b, i, S)$ condition ensures this: it requires that a quorum of *rejecters* have promised, by voting on a branch higher than b, to never accept $\langle b, i\rangle$.

We simplified Ferry's SUPs by relying on the following integrity invariant for individual node states S, which we verified to be inductive over SUP-constrained updates:

$$\begin{aligned}(\forall b, N, i. \quad S.\texttt{accepts}[\langle b, N\rangle] = i \implies \langle b, i\rangle \in S.\texttt{tree}) \\ \wedge \ (\forall \langle b, i\rangle \in S.\texttt{tree}. \quad \langle b, i\rangle \sqsubseteq_{S.\texttt{tree}} \texttt{head}(S.\texttt{tree}) \ \vee \ \texttt{deferred}(b, i, S)).\end{aligned}$$

This condition states that every accept-record corresponds to an existing entry, and every entry bypassed by the current highest branch has been deferred. Finally, we relied on the following assumption for configured vote-quorum and accept-quorum sizes, which ensures that every vote-quorum intersects with every accept-quorum, and that every pair of vote-quorums intersect:

$$|S.\texttt{voters}| < (S.\texttt{voteQ} + S.\texttt{acceptQ}) \ \wedge \ |S.\texttt{voters}| < (S.\texttt{voteQ} + S.\texttt{voteQ}).$$

The symbolic representation of Ferry's action functions and effect-handler function was derived automatically from the implementation by Super-V.

Decidability. When instantiated with the symbolic representation of Ferry's code and with the manually-discovered SUP artifacts, the resulting verification conditions for consensus safety fall within *Extended EPR*: a decidable fragment of logic. This means that we were able to dispatch them directly to an SMT solver (CVC5 [Barbosa et al. 2022]), and could expect the solver to reliably verify or falsify them—a crucial feature during the trial-and-error process of identifying the SUPs.

We were able to achieve this immediate decidability because our verification conditions did not range over complex global system states and execution histories—only over pairs of node states and updates. Ferry's node states themselves contain complex structures—maps and trees—but Ferry uses restricted interfaces to these structures which could be axiomatized without creating sort cycles. This would not have been possible if we had to accommodate sort edges that were mandated by the verification condition templates themselves.

In contrast, existing projects that have verified Raft using the traditional verification approach—induction over global system states—have required additional manual effort to overcome the undecidability of their initial verification conditions. The approach of Woos et al. [2016] eschewed automation entirely, requiring the authors to write an immense proof consisting of 50,000 lines of Coq code. The authors of Taube et al. [2018] recovered decidable automation by manually developing an abstraction of the global system and then verifying relations between this abstraction and the components of the concrete system. As a result, they verified their implementation of Raft using 300 lines of IVy code to define the necessary invariants and manage the ghost-state. To verify Ferry, we used 110 lines of code in Super-V's DSL to define the necessary specification and artifacts. Verifying Ferry takes about 50min using a 2019 Intel Core-i5 laptop.

5.4 Communication Complexity

Ferry's design gives it a communication complexity that is different from Raft and Paxos. For a leader election in Raft, $2(n-1)$ messages are sent for n nodes: the candidate node sends a `RequestVote` message to each of its $n-1$ peers, and each peer sends a response. In Ferry, $n + (n-1)^2$ messages are sent: the candidate broadcasts a `Vote` update to its $n-1$ peers, and each peer broadcasts its own `Vote` to its $n-1$ peers. These numbers repeat for the `Propose` update its `Accepts`. In practice, the impact of higher complexity is limited: consensus clusters typically operate 3 or 5 nodes.

Replicated state gives Ferry an advantage when considering communication *delays*. For Raft, 3 communication delays must pass for the followers to learn that a newly-proposed entry has been committed: one delay for the leader to broadcast its `AppendEntries` messages to its followers, one delay for those followers to send their response messages to the leader, and one delay for the leader to send a commit notification back to the followers (piggy-backed on the next `AppendEntries` request). In contrast, Ferry nodes learn of a committed entry after only 2 communication delays: one delay for the leader to broadcast its `Propose` update, and one delay for the followers to broadcast their `Accept` updates—to the leader and to one another.

5.5 Performance Evaluation

We sought to evaluate Ferry—as a representative of the general class of consensus algorithms built on weakly-replicated state—against algorithms based on standard practice. We had two research questions. **RQ1**: Does Ferry perform similarly to standard-practice consensus algorithms under normal network conditions? **RQ2**: Under failure conditions, is Ferry's ability to recover similar to that of standard-practice consensus algorithm?

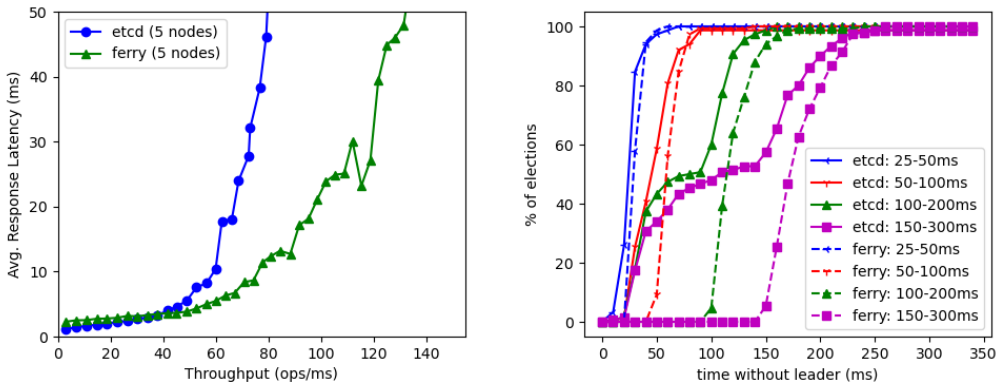


Fig. 9. Throughput and fault recovery experiments, comparing Ferry to existing Raft implementations.

To answer **RQ1** we deployed two log consensus clusters in a typical deployment environment: clusters of 5 machines (Intel Xeon Platinum 8380s, 2.30Ghz) co-located in a single datacenter with an average round-trip communication delay of 0.04ms. One cluster ran Ferry, and the other ran `etcd`: a modern, widely used, replicated key-value store that uses Raft to enforce strong consistency [etcd 2024]. Ten client processes on a separate machine made repeated PUT requests (no GETs) of each cluster. Our hypothesis was that Ferry's higher message complexity would give Ferry a small latency overhead, compared to `etcd`, that would remain constant as the request rate increased.

The results of this experiment, plotted in Fig. 9a, support our hypothesis on low request rates—Ferry starts with a 1.2ms latency disadvantage—but on high request rates it outperforms `etcd`, against our expectations. We do not take these results to suggest that Ferry's design provides any fundamental advantage over that of `etcd`, but we do take them as a positive answer to **RQ1**: Ferry's replicated state basis does not impose any significant performance penalty.

For **RQ2**, we evaluated Ferry's ability to recover quickly from leader failure. In this experiment, which mirrors an experiment presented in the original Raft paper [Ongaro and Ousterhout 2014], we varied the range of random timeouts that trigger leader elections. For each time range, we ran 150 elections—for each, we killed the leader node of a cluster and then measured the time it took for another node to notify the client that it had assumed leadership. Our results (Fig. 9b) are similar to those reported in the Raft paper, answering **RQ2** positively: like Raft, Ferry's failure recovery speed is a function of the timeout ranges used. Given the same timeout configuration parameters, Ferry and etcd have similar 90th-percentile election delays, though Ferry's delays are longer (about 10% longer for 150-300ms). However, a smaller fraction of etcd elections are triggered and completed much more quickly; etcd has an unexpected two-phase curve that is not shared by Ferry or Ongaro and Ousterhout [2014]. This is probably not desirable behavior: if election timeouts trigger significantly faster than their configured interval, followers may launch spurious elections (with associated delays) in between the heartbeats of a functioning leader.

6 Related Work

As implied in Sec. 1, there are two broad categories of related work: verification approaches that focus on strong replication *implementations* (i.e. consensus algorithms) and those that focus on replicated state *applications*. Notably, there does not exist any prior work that intersects these two categories as we do: verification of distributed consensus algorithms *as* replicated state applications.

Verification of Consensus Algorithms. IronFleet [Hawblitzel et al. 2015] is a verification methodology for distributed programs based on gradual refinement of specifications to low-level implementations. In contrast, our verification focuses on high-level implementations of protocols while trusting a replicated state library/service to correctly handle the low-level concerns. While IronFleet's verification extends to liveness properties, we focus only on safety. Verdi [Wilcox et al. 2015] is a Coq-based verification framework for distributed systems that focuses on abstracting layered fault models, using *transformers*. It has been used to verify Raft [Ongaro and Ousterhout 2014], showing that it acts as a transformer with a linearizability guarantee. Verification is primarily driven by inductive invariants and proofs are tactic-based with little automation. Consequently, the proof effort is often prohibitive: Raft verification required a 50,000-line proof [Woos et al. 2016]. Our approach presents a light-weight alternative that is more amenable to automation. Ivy [Padon et al. 2016; Taube et al. 2018] is a verification tool that focuses on achieving predictable automation by modularizing implementations such that each module is limited to decidable logic. Their verification methodology requires identification of system-wide inductive invariants, which we avoid.

Pretend Synchrony [v. Gleissenthall et al. 2019] is a recently proposed verification methodology for distributed protocol implementations with the express purpose of reducing the size of inductive invariants. Their proof technique computes a sound synchronization of the protocol, which can be verified as a sequential program without complex invariants. The synchronization is however conditional. In particular, their SPCP programming model severely restricts message-broadcast loops over replica IDs and does not admit the Raft/Multi-Paxos behavior of using one leader election to drive many successive log proposals.

Verification of Replicated Applications. Gotsman et al. [2016] present a rely-guarantee based verification framework for replicated state applications. Balegas et al. [2015] describe a static analysis that discovers concurrency-related bugs in a replicated application and identifies which can be fixed by strengthening consistency. In both cases, strong consistency is axiomatized to reason about the safety of applications. In contrast, we reason about the safety of distributed protocols that implement strong consistency. Bloom [Alvaro et al. 2011] presents a static analysis to identify coordination points in distributed programs. Bloom is based on the CALM theorem [Ameloot et al. 2013], which observes that logically monotonic programs do not need coordination to achieve consistency. Our approach is based on similar intuitions about monotonicity, although we exclusively focus on verifying programs that perform coordination. Several authors have proposed verification frameworks for CRDTs [De Porre et al. 2023; Gomes et al. 2017; Liu et al. 2020; Zakhour et al. 2023] which satisfy *strong eventual consistency*, only requiring the replicas to *eventually agree*. In contrast, our approach verifies consensus, which insists that replicas *never conflict*.

Augmenting the Network Model. In existing work, the *unreliable fault detector* [Chandra and Toueg 1996; Gafni 1998] network model abstraction has been used to ease the verification of *liveness/termination* for consensus algorithms. This approach is orthogonal to our own, which augments the network model using the causal delivery assumption to ease the verification of *safety* for consensus algorithms. In practice, Ferry (like Raft) uses election-triggering timeouts to achieve the probabilistic progress measured in Sec. 5.5. However, we do not address verification of liveness, and so we have not formalized this timeout mechanism as a unreliable fault detector.

Data-Availability Statement

Our artifact for this paper includes the Super-V verification and runtime system, as well as our implementation of the Ferry consensus algorithm, which uses Super-V's embedded language.

Acknowledgments

Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation.

References

Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. 2019. Dissecting the Performance of Strongly-Consistent Replication Protocols. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1696–1710. https://doi.org/10.1145/3299869.3319893

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach. In *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. www.cidrdb.org, 249–260. http://cidrdb.org/cidr2011/Papers/CIDR11_Paper35.pdf

Tom J. Ameloot, Frank Neven, and Jan Van Den Bussche. 2013. Relational Transducers for Declarative Networking. *J. ACM* 60, 2, Article 15 (may 2013), 38 pages. doi:10.1145/2450142.2450151

Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. ACM, New York, NY, USA, 761–772. doi:10.1145/2463676.2465279

Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. 2015. Putting Consistency Back into Eventual Consistency. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. ACM, New York, NY, USA, Article 6, 16 pages. doi:10.1145/2741948.2741972

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. doi:10.1007/978-3-030-99524-9_24

Russell Brown, Sean Cribbs, Christopher Meiklejohn, and Sam Elliott. 2014. Riak DT Map: A Composable, Convergent Replicated Dictionary. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency* (Amsterdam, The Netherlands) *(PaPEC '14)*. ACM, New York, NY, USA, Article 1, 1 pages. doi:10.1145/2596631.2596633

Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable failure detectors for reliable distributed systems. *J. ACM* 43, 2 (March 1996), 225–267. doi:10.1145/226643.226647

Confluent. 2024. Kafka Raft Metadata Mode. https://docs.confluent.io/platform/current/kafka-metadata/kraft.html.

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (aug 2013), 22 pages. doi:10.1145/2491245

Kevin De Porre, Carla Ferreira, and Elisa Gonzalez Boix. 2023. VeriFx: Correct Replicated Data Types for the Masses. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:45. doi:10.4230/LIPIcs.ECOOP.2023.9

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 205–220. doi:10.1145/1294261.1294281

etcd. 2024. etcd: a distributed reliable key-value store. https://github.com/etcd-io/etcd. Accessed: 2024-10-15.

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

Eli Gafni. 1998. Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing* (Puerto Vallarta, Mexico) *(PODC '98)*. Association for Computing Machinery, New York, NY, USA, 143–152. doi:10.1145/277697.277724

Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News* 33, 2 (June 2002), 51–59. doi:10.1145/564585.564601

Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. 2017. Verifying strong eventual consistency in distributed systems. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 109 (oct 2017), 28 pages. doi:10.1145/

3133933

Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'M Strong Enough: Reasoning About Consistency Choices in Distributed Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. ACM, New York, NY, USA, 371–384. doi:10.1145/2837614.2837625

Tobias Grieger. 2019. Joint Consensus in Raft. https://www.cockroachlabs.com/blog/joint-consensus-raft/. Accessed: March 1, 2025.

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 *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. ACM, New York, NY, USA, 1–17. doi:10.1145/2815400.2815428

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. doi:10.1145/78969.78972

Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2017. Flexible Paxos: Quorum Intersection Revisited. In *20th International Conference on Principles of Distributed Systems (OPODIS 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 70)*, Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 25:1–25:14. doi:10.4230/LIPIcs.OPODIS.2016.25

Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN '11)*. IEEE Computer Society, USA, 245–256. doi:10.1109/DSN.2011.5958223

Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. 2020. Refinement for Structured Concurrent Programs. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I* (Los Angeles, CA, USA). Springer-Verlag, Berlin, Heidelberg, 275–298. doi:10.1007/978-3-030-53288-8_14

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

Leslie Lamport. 2004. *Generalized Consensus and Paxos.* Microsoft Research. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2005-33.pdf

Leslie Lamport. 2006. Fast Paxos. *Distrib. Comput.* 19, 2 (oct 2006), 79–103. doi:10.1007/s00446-006-0005-x

Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. 2020. Verifying Replicated Data Types with Typeclass Refinements in Liquid Haskell. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 216 (nov 2020), 30 pages. doi:10.1145/3428284

Nancy A. Lynch. 1996. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Dmitry Martyanov. 2020. CRDT in Production. https://www.infoq.com/presentations/crdt-production/. Accessed: March 1, 2025.

Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is more consensus in Egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 358–372. doi:10.1145/2517349.2517350

Kartik Nagar and Suresh Jagannathan. 2019. Automated Parameterized Verification of CRDTs. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 459–477.

Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 305–320.

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. doi:10.1145/3140568

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

Patrick Redmond, Gan Shen, Niki Vazou, and Lindsey Kuper. 2023. Verified Causal Broadcast with Liquid Haskell. In *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages* (<conf-loc>, <city>Copenhagen</city>, <country>Denmark</country>, </conf-loc>) *(IFL '22)*. Association for Computing Machinery, New York, NY, USA, Article 6, 13 pages. doi:10.1145/3587216.3587222

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

William Schultz, Siyuan Zhou, and Stavros Tripakis. 2021. Brief Announcement: Design and Verification of a Logless Dynamic Reconfiguration Protocol in MongoDB Replication. In *35th International Symposium on Distributed Computing (DISC 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 209)*, Seth Gilbert (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 61:1–61:4. doi:10.4230/LIPIcs.DISC.2021.61

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. doi:10.1145/3158116

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems* (Grenoble, France) *(SSS'11)*. Springer-Verlag, Berlin, Heidelberg, 386–400. http://dl.acm.org/citation.cfm?id=2050613.2050642

KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 413–424. doi:10.1145/2737924.2737981

Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. 2018. Modularity for Decidability of Deductive Verification with Applications to Distributed Systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 662–677. doi:10.1145/3192366.3192414

Pedro Teixeira. 2017. Decentralized Real-Time Collaborative Documents - Conflict-free editing in the browser using js-ipfs and CRDTs. https://ipfs.io/blog/30-js-ipfs-crdts.md.

Klaus v. Gleissenthall, Rami Gökhan Kıcı, 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. doi:10.1145/3290372

Evan Wallace. 2019. How Figma's Multiplayer Technology Works. https://www.figma.com/blog/how-figmas-multiplayer-technology-works/. Accessed: March 1, 2025.

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 *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 357–368. doi:10.1145/2737924.2737958

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

George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2023. Type-Checking CRDT Convergence. *Proc. ACM Program. Lang.* 7, PLDI, Article 162 (jun 2023), 24 pages. doi:10.1145/3591276

Received 2024-10-16; accepted 2025-02-18