Distributed Consensus Algorithms as Replicated State Applications

Nicholas V. Lewchenko  
University of Colorado Boulder  
USA  
nicholas.lewchenko@colorado.edu

Gowtham Kaki  
University of Colorado Boulder  
USA  
Gowtham.Kaki@colorado.edu

Abstract
To verify implementations of distributed consensus algorithms, such as Raft and Paxos, developers must identify and express complex inductive invariants. Much of this complexity is due to explicit message-passing—the states of nodes and the history of messages they have sent must be exhaustively correlated.

In this paper, we show that verification artifacts can be simplified by implementing consensus algorithms in a weakly-consistent replicated state model that omits explicit message-passing. Based on this model, we define a novel proof theory based on interference contracts and find that distributed log consensus is partly a consequence of eventual consistency in this setting. We use our programming model and proof theory to explore the implementation and verification of the Raft consensus protocol.

1 Introduction
Distributed consensus algorithms allow nodes in a distributed system to agree on actions—such as committing a bank transaction—without expensive synchronous communication [6, 8]. Implementing distributed consensus is an error-prone process. Bugs that arise from concurrent behavior are hard to intuitively predict, and testing is of limited use in the highly non-deterministic network setting. For these reasons, formal verification for distributed consensus implementations has been a project of great interest [5, 10, 12, 13].

Unfortunately, consensus implementations are traditionally written using explicit, asynchronous message-passing, which adds work to the verification process. In the standard transition system verification approach, messages are tracked as ghost state, which must be painstakingly constrained by inductive invariants to conform with the concrete states of system nodes [12, 13]. If automated verification is required, this ghost state must be creatively designed to avoid undecidable logic fragments [10].

In this paper, we explore the implementation and verification of distributed consensus algorithms using replicated state, a programming model that excludes explicit message-passing. Typically, consensus algorithms are used as an underlying system to support a strongly-consistent replicated state environment for higher-level application code. We take

replicated state further by showing that consensus algorithms themselves can be efficiently implemented in a weakly-consistent replicated state model—and that those implementations are easier to verify than traditional message-passing equivalents.

To do so, we define the Identity-aware, Causally-consistent Replicated State (ICRS) programming model and a novel proof theory for verifying properties of ICRS applications. We explain how our proof theory expresses distributed log consensus in a new way, by leveraging generic eventual consistency, and we apply these concepts to a work-in-progress implementation and verification of the Raft consensus protocol.

2 Example: Leader Election
In this section, we explain the key details of our programming and verification approach using leader election as an example application. Leader election, a standard component of consensus algorithms, consists of nodes voting for a leader to perform some role. Complete consensus algorithms elect leaders for a series of terms, avoiding tied-election deadlock, but for simplicity our example will only concern safety for a single election term.

2.1 Leader Election Algorithm in Replicated State
Fig. 1 illustrates an execution of our simple leader election algorithm. Each participating node maintains a set of votes (for itself and its peers) as local state. When node $r_1$ votes for
itself, it adds the vote \( \langle r_1, r_1 \rangle \) to its state, and also broadcasts the vote to \( r_2 \), causing \( r_2 \) to add the vote to its own state.

In general, a replicated state application consists of nodes—which we call replicas—that each store a complete copy of the application state. When a replica updates its state copy, it reports this to its peers via broadcast message, causing them to make equivalent changes to their copies. Eventual consistency is a common property that requires replica states to match whenever all messages have been delivered. The execution in Fig. 1 exhibits eventual consistency: the states of \( r_1 \) and \( r_2 \) match at the beginning (when no messages have been sent) and the end (when both the \( e_1 \) and \( e_2 \) messages have been delivered).

When a leader election replica’s state contains a quorum of votes for a single candidate (itself or a peer), it considers that candidate to be the leader. We formalize this as the following predicate on states \( s \), using a preconfigured majority size \( Q \).

\[
\text{Leader}(s, r) \iff \\
\text{Quorum}(s, r) \land \neg \text{DoubleQuorum}(s)
\]

\[
\text{Quorum}(s, r) \iff \\
\{ \langle r_1, r \rangle \in s \} \geq Q
\]

\[
\text{DoubleQuorum}(s) \iff \\
\exists r_1, r_2. \ r_1 \neq r_2 \land \text{Quorum}(s, r_1) \land \text{Quorum}(s, r_2)
\]

Note that a candidate is only recognized as the leader when no quorums for other candidates have been witnessed. Therefore, by definition:

\[
r_1 \neq r_2 \implies \neg \text{Leader}(s, r_1) \lor \neg \text{Leader}(s, r_2).
\]

The execution in Fig. 1 concludes with both replica states recognizing \( r_2 \) as the leader:

\[
\text{Leader}(\{\langle r_1, r_2 \rangle, \langle r_2, r_2 \rangle\}, r_2).
\]

### 2.2 Agreement from Single-Replica Safety Property

Before we consider the implementation of this leader election algorithm, we formalize our required safety properties.

The primary safety property for leader election is agreement: if one replica sees \( r_1 \) as the leader, no other replica should see a different candidate \( r_2 \) as the leader. This property is an invariant on the whole network state, rather than a single replica’s state. Our verification approach avoids reasoning directly about whole-network invariants like this—instead, we will indirectly ensure agreement as a consequence of single-replica finality and generic eventual consistency for replicated applications.

By finality, we mean the property that election outcomes cannot be revoked or invalidated. This is not a whole-network invariant like agreement—rather, it is a temporal property on individual replica states. We formalize finality as \( I_f \), a preorder on replica states:

\[
\langle s_1, s_2 \rangle \in I_f \iff \\
\forall r. \ \text{Leader}(s_1, r) \implies \text{Leader}(s_2, r)
\]

If each replica’s state monotonically increases according to \( I_f \), then the complete network maintains finality—no witnessed leader is forgotten. We call \( I_f \) a replica trace invariant when it is satisfied by an execution in this way. Our example execution in Fig. 1 satisfies \( I_f \): neither replica transitions to a non-leader state after witnessing the quorum for \( r_2 \).

Notice that our example execution also satisfies agreement—in fact, this is guaranteed since it satisfies both finality and eventual consistency. Why? Consider two replicas that break agreement, by witnessing two different leaders. If both replicas maintain finality—their states continue to witness the same divergent leaders—then their states cannot possibly match in the future, contradicting eventual consistency.

Therefore, we can safely focus our leader election verification task on two goals: finality as represented by \( I_f \), and generic eventual consistency.

In Sec. 5, we will show how distributed log consensus—the goal of complete consensus algorithms—can be similarly captured as a consequence of eventual consistency and log-finality.

### 2.3 Implementation by Replicated State Update

The core of our leader election implementation—and the sole object of verification—is the voteFor\( (r) \) transaction, invoked by a replica in order to cast a vote for \( r \). We leave the safety-irrelevant detail of choosing candidates abstract.

\[
\text{voteFor}(r) := \lambda (\text{self}). \ \lambda (\text{state}). \quad (1)
\]

\[
\text{assert} \ \text{nonVoter}(\text{self}, \text{state}). \quad (2)
\]

\[
\text{update} \ \text{Insert}(\langle \text{self}, r \rangle). \quad (3)
\]

\[
\text{nonVoter}(r,s) \iff \forall r_1. \ \langle r, r_1 \rangle \notin s
\]

\[
\text{Insert}(v) := \lambda s. \ (s \cup \{v\})
\]

Once invoked, this transaction receives two runtime-provided arguments (line 2): the invoking replica’s ID \( \text{self} \), and its local state copy \( \text{state} \). This access to a unique replica ID is the identity-aware feature of our ICRS model, used here to prevent double-voting.

On line 3, the local replica state is checked, asserting that the voting replica has not previously cast a vote (and safely aborting the transaction if the assertion fails).

Finally line 4 updates the replicated state by adding the new vote for \( r \), cast by \( \text{self} \). More precisely, this statement performs three actions that a developer would need to implement individually in the traditional approach:

1. Insert the vote \( \langle \text{self}, r \rangle \) into the local state of \( \text{self} \).
2. Broadcast a message to \( \text{self} \)'s peers, announcing that it has voted for \( r \).
3. Run a handler on each peer that receives the message, which adds $\langle \text{self}, r \rangle$ to their local states.

This unification of three operational details into a single program statement (update) is the key advantage of our election implementation, from a verification standpoint. Verification for the traditional message-passing equivalent involves proving that a vote message is only sent when the voting node has actually added the vote to its own state, and likewise that a receiving node only adds a vote to its own state in response to a matching vote message [10, 13]. In the replicated state programming model, the model itself enforces this correspondence, reducing the number of details that must be expressed and verified by the developer.

2.4 Verification by Interference Contract

At a high level, maintaining leader election finality depends on not double-voting. Casting two different votes in the name of a single replica may create a double-quorum, which would nullify the Leader$(s, r)$ predicate, and thus violate $I_f$.

Can voteFor$(r)$ create a double-vote? When $r_1$ executes voteFor$(r)$, it asserts that there are no preexisting votes from $r_1$ in its local state. But what if another replica $r_2$ voted in $r_1$’s name? Then, when $r_2$ receives $r_1$’s legitimate vote, the state of $r_2$ could contain a double-vote. Fig. 2 illustrates this scenario, in which $I_f$ is violated by $r_2$’s trace.

Of course, we know that $r_2$ cannot actually vote in $r_1$’s name—the voteFor$(r)$ transaction only votes using self. We capture this intuition for verification purposes by defining the following interference contract $C_e$, a parameterized preorder which generalizes the notion of an inductive invariant and serves a similar purpose.

\[
\langle s_1, s_2 \rangle \in C_e(r) \iff \\
(\neg \text{DoubleVote}(s_1) \implies \neg \text{DoubleVote}(s_2)) \\
\land (\text{nonVoter}(r, s_1) \implies \text{nonVoter}(r, s_2))
\]

DoubleVote$(s) \iff \\
\exists r_1, r_2, r_3. \ r_2 \neq r_3 \land \langle r_1, r_2 \rangle \in s \land \langle r_1, r_3 \rangle \in s
\]

An interference contract, $C_e$, constrains the state changes that can be made concurrently to a named replica’s transactions. When replica $r_1$ performs a transaction on local state $s_1$, it is guaranteed that every state $s_2$ that its update is applied to—including both $r_1$’s local state and the states of $r_1$’s remote peers—will satisfy $\langle s_1, s_2 \rangle \in C_e(r_1)$.

Essentially, $C_e$ is a rely-condition that is parameterized by replica IDs, in the sense of rely/guarantee reasoning.

$C_e$ disallows double-voting by any replica, and voting by any replica using another replica’s ID. In Fig. 2, we can see that $\neg C_e(r_1, s_1, s_2)$—therefore, we can dismiss Fig. 2 as a counterexample to safety of voteFor$(r)$.

Like an inductive invariant, our interference contract imposes more verification goals on our application: we must show that $r_1$ preserves $C_e(r_2)$, for any other replica $r_2$, in addition to $I_f$. But it also gives us stronger assumptions for proving those goals: we may assume that concurrent updates preserve $C_e(r_1)$.

For example, the following proof sketch shows that, when $r_1$ executes $\text{voteFor}(r)$, the resulting update does not double-vote when applied to a remote state $s_2$, as required by $C_e(r_2)$:

1. The assert in $\text{voteFor}(r_1)$ gives $\text{nonVoter}(r_1, \text{state})$.
2. $\text{nonVoter}(r_1, \text{state}) \land \langle \text{state}, s_2 \rangle \in C_e(r_1)$ implies that $\neg \text{nonVoter}(r_1, s_2)$.
3. $s_2$ contains no votes cast by $r_1$, thus the post-update state $s_2 \cup \langle r_1, r \rangle$ contains only one vote cast by $r_1$.

We can extend the proof sketch to show that $I_f$ is also preserved by relying on the interference contract. $I_f$ is only violated if $\text{Leader}(s_2, r_1)$, for some candidate $r_1$, and $\neg \text{Leader}(s_2 \cup \langle r_1, r \rangle, r_1)$. This could be the case if the $\langle r_1, r \rangle$ vote creates a double-quorum. However, a double-quorum requires some replica to have voted twice, and we have shown that $s_2 \cup \langle r_1, r \rangle$ does not introduce a double-vote.

By avoiding explicit message-passing (fusing it with local state updates), we have reduced the size of the necessary verification artifacts. The $C_e$ interference contract is smaller than the inductive invariant needed for an equivalent message-passing implementation.

Finally, eventual consistency for leader election does not depend on the interference contract—it follows from the fact that all updates are set-insertions, which perfectly commute with one another. Combining eventual consistency with the $I_f$ trace invariant ensures our desired leader election agreement property.

3 The ICRS Programming Model

Replicated state programming models have been formalized and implemented to provide a variety of guarantees suitable to a variety of applications. Many models provide configurable strong consistency guarantees, enforced at runtime by blocking coordination [2, 4, 7, 9].

Our goal is to implement consensus algorithms with runtime performance comparable to existing non-replicated implementations. Therefore, our programming model, identity-aware, causally-consistent replicated state (ICRS), is limited to non-blocking consistency guarantees.

3.1 Consistency Guarantees

The first consistency guarantee of ICRS, identity-awareness, allows application code running on a replica to access a replica ID that is guaranteed to be unique—the same replica ID will not be given to another replica’s application code. As a form of unique ID service, this feature represents a lightweight coordination mechanism that can be implemented in a non-blocking manner (by pre-configuring replicas with static IDs) [2].
We assume that ICRS applications only produce replica traces in executions that satisfy causal consistency.

Identity awareness is crucial to consensus algorithm implementations. As demonstrated in Sec. 2.4, quorum-based reasoning depends on replica ID uniqueness.

We formally model executions of ICRS applications as with the leader election property that no state ever witnesses a double-vote can be captured as a replica trace invariant:

Definition 3.2 (Eventual consistency). An execution $X$ is eventually consistent iff every trace’s state sequence ends with the same state.

3.3 Replica Trace Invariant

We represent application-specific safety properties of ICRS applications as state-preorders called replica trace invariants. Given a replica trace invariant $I$, a replica that sees state $s_1$ is guaranteed to only see $s_2$ in the future if $(s_1, s_2) \in I$.

Definition 3.3 (Replica trace invariant). An execution $X$ satisfies state-preorder $I$ as a replica trace invariant iff every replica trace in $X$ has a state sequence that monotonically increases according to $I$.

Replica trace invariants are temporal properties, but they generalize non-temporal single-state invariants. For example, the leader election property that no state ever witnesses a double-vote can be captured as a replica trace invariant:

$$\{ s_1, s_2 \mid \neg \text{DoubleVote}(s_1) \implies \neg \text{DoubleVote}(s_2) \}.$$
D is the set of all pairs of distinct replica IDs:
\[ \forall s_1, s_2 \in S. \ \forall (r_1, r_2) \in D. \ (T(r_1, s_1) \leadsto u \land (s_1, s_2) \in C(r_1)) \implies (s_2, u(s_2)) \in (C(r_2) \cap I) \]
\[ C \vdash \text{Safe}(T, I) \]

The notation \( T(r_1, s_1) \leadsto u \) means that transaction \( T \), when self is set to \( r_1 \) and state is set to \( s_1 \), passes all assertions and issues state update \( u \in S \rightarrow S \). In this rule, \( r_1 \) is the transaction’s local replica, and \( r_2 \) is any remote replica. Therefore, we may rely on \( C(r_1) \), and we must guarantee \( C(r_2) \) in addition to \( I \). Note that \( s_2 \) in the rule represents both \( r_1 \)'s local state (when \( s_2 = s_1 \)) and the states of \( r_1 \)'s remote peers.

**Theorem 4.1** (Verification soundness). *Given causally consistent execution \( X \) generated by application \( \{T_1, \ldots, T_n\} \), and interference contract \( C \) such that \( C \vdash \text{Safe}(T_1, I) \land \cdots \land C \vdash \text{Safe}(T_n, I) \), it is the case that \( X \) satisfies \( I \) as a replica trace invariant.*

5. **Case Study: Raft**

Raft [8] is a widely used consensus protocol that, like its older variant Paxos [6], has been a target of many formal verification efforts [10, 11, 13]. One such effort, using the Verdi framework, required discovering and proving 90 system invariants [13].

Our goal is to verify an implementation of the Raft protocol while reducing the size of the necessary verification artifacts, by restricting the implementation to the ICRS programming model. This effort is a work-in-progress; while we have an informal proof, the exhaustive formalization is still pending.

The original presentation of Raft uses a remote procedure call model, in which nodes update their local states in step with synchronous call-and-response message exchanges between pairs of nodes. Creative design work was needed to adapt this protocol to the replicated state model. The result is a replicated state with four components, acted on by three transactions.

5.1 **State Model**

The state has the following components:

- \( \text{votes} \in \mathcal{P}(\text{Rid} \times \text{Rid} \times \text{Nat}) \) The set of votes. An element \( (r_0, r_c, t) \) represents \( r_0 \) voting for \( r_c \) to be leader of \( t \). A replica \( r \) becomes the leader for term \( t \) when a quorum of votes for \( r \) is present.
- \( \text{term} \in \text{Nat} \) The latest term in which the log has been modified.
- \( \text{accepts} \in \mathcal{P}(\text{Rid} \times \text{Nat} \times \text{Nat}) \) A record of what prefix of the log has been accepted (i.e., witnessed) by each replica in each term. The entry \( (r, t, i) \) means that \( r \) has accepted the \( i \)-length prefix of the log written in term \( t \).

1\( \log \in \log \) The current proposed log. The accepts set determines which prefix of this log is committed. The rest is subject to change.

As in Sec. 2, we use \( Q \) to denote the quorum size for the set of participating replica IDs.

There are two key predicates on the state. \( \text{Leader}(s, r, t) \) states that a given replica ID is the (only) elected leader for a term \( t \)—the definition is similar to that of \( \text{Leader}(s, r) \) from Sec. 2. \( \text{Committed} \) determines whether the given index is committed in the given term, according to the given state. When an index is committed in one term, it is also considered committed in all later terms.

\[ \exists l \in \text{Nat}. \ t \leq t \land t \land |\{r | \exists l_1, i \leq l_1 \land (r, t, l_1) \in s.\text{accepts} \}| \geq Q \]

5.2 **Distributed Log Consensus**

Our key safety condition \( l_c \), corresponding to Raft’s state machine safety property, requires that a log index, once witnessed as committed, does not change in the future.

\[ \forall i \in \text{Nat}. \forall t \in \text{Nat}. \ \exists l \in \text{Nat}. \ t \leq t \implies \text{PrefixMatch}(i, s, \log, s, \log) \]

Like our example in Sec. 2, the \( l_c \) trace invariant only explicitly represents the finality aspect of log consensus, but implicitly guarantees agreement between replicas when eventual consistency is maintained.

5.3 **Implementation**

Fig. 3 defines the transactions and updates of our replicated state Raft implementation. The \( \text{vote}(r, t) \) transaction is analogous to the \( \text{voteFor}(r) \) transaction in Sec. 2, but in this case leaders are elected for particular terms.

Elected leaders use the \( \text{propose}(l, t) \) transaction to propose new log entries, tagged with their elected term. We assert that the local replica is indeed the elected leader for \( t \), and that the log it proposes is strictly an extension of the log it has so-far seen.

When these assertions hold, \( \text{propose}(l, t) \) issues the update \( \text{NewLog}(l, t) \), which sets the log to \( l \) and the term to \( t \)—but only on replica states where the term has not already increased beyond \( t \). Note that two \( \text{NewLog} \) updates for the same term would not commute with each other. To verify that eventual consistency is still maintained, we must ensure that two such updates never actually occur concurrently in our application (Sec. 5.4).

The \( \text{accept}(i) \) transaction is used by replica to announce that it has seen the \( i \)-length prefix of the log in the replica’s current term. This action is only allowed when the replica has not already voted in any greater term. When a quorum
vote(r, t) :=
\lambda(self), \lambda(state).
assert nonVoter(state, self, t).
update votes.Insert((self, r, t))

propose(l, t) :=
\lambda(self), \lambda(state).
assert Leader(state, self, t).
assert logPrefix(state.log, l).
update NewLog(l, t)

accept(i) :=
\lambda(self), \lambda(state).
assert nonVoterOver(state, self, state.term)
assert HasSize(state.log, i).
update accepts.Insert((state, term, i))

NewLog(l, t) := \lambda s.
case (t \geq s.term) \rightarrow s\{term = t, log = l\}
case (t < s.term) \rightarrow s

\textbf{Figure 3.} Transactions and updates for the Raft implementation.

of accepts for a single term meet or exceed an index i, that index is automatically recognized by any observing replica as committed.

\textbf{5.4 Verifying Eventual Consistency}

In cases where updates do not universally commute, as with the NewLog(l, t) update, we verify eventual consistency by providing an \textit{update guard}: a single-state predicate parameterized by updates:

\[
G(\text{NewLog}(l, t)) \iff \text{term} \neq t \lor \text{logPrefix}(s, \text{log}, l)
\]

The update guard represents an assertion that every state the given update can encounter will satisfy the given property. We will not check this condition at runtime—rather, we statically verify that it is a consequence of our interference contract, just like the replica trace invariant.

We are able to verify \(G\) in this way because only one replica—the elected leader—ever proposes for a given term, and a single replica’s updates are not concurrent to each other. This allows us to ignore the non-commuting pair of NewLog(l, t) and NewLog(l, t), where \(l_1 \neq l_2\).

\[
C_1(r, s_1, s_2) \iff \text{(Vote Safety)}
\]

\[
\text{VoteFreeze}(r, s_1.votes, s_2.votes) \wedge \neg\text{DoubleVote}(s_1.votes) \implies \neg\text{DoubleVote}(s_2.votes)
\]

\[
C_2(r, s_1, s_2) \iff \text{(Accept Safety)}
\]

\[
\text{AcceptFreeze}(r, s_1.accepts, s_2.accepts) \wedge \text{VoterRestrict}(s_1.votes, s_1.accepts, s_2.accepts) \wedge s_1.accepts \subseteq s_2.accepts \wedge \text{AcceptTerm}(s_1.accepts, s_1.term) \implies \text{AcceptTerm}(s_2.accepts, s_2.term)
\]

\[
C_3(r, s_1, s_2) \iff \text{(Log Safety)}
\]

\[
\text{LiveMatch}(s_2.votes, s_2.accepts, s_1.term, s_1.log, s_2.log) \wedge s_1.term \leq s_2.term \wedge \text{LiveBound}(s_1.votes, s_1.accepts, s_1.log) \implies \text{LiveBound}(s_2.votes, s_2.accepts, s_2.log)
\]

\[
C_4(r, s_1, s_2) \iff \text{(Leader Log)}
\]

\[
\text{Leader}(s_1, r, s_2) \implies s_1.term = s_2.term \wedge s_1.log = s_2.log
\]

\textbf{Figure 4.} Interference contract for verifying that the Raft implementation satisfies \(I_e\) as a replica trace invariant.

\textbf{5.5 Verifying Distributed Log Consensus}

The interference contract for verifying that Raft satisfies the \(I_e\) trace invariant is shown in Fig 4.

The \text{VoteFreeze}(r, v_1, v_2) and \text{AcceptFreeze}(r, m_1, m_2) rules express that remote replicas will not vote or accept, respectively, using \(r\)'s ID. The \text{VoterRestrict}(r, m_1, m_2) rule in \(C_2\) expresses the rule that a replica cannot accept on a \(t_1\) when it has already voted in a greater term \(t_2\) (as ensured by the \text{nonVoterOver} assertion in \text{accept}(i)). \text{AcceptTerm} forbids entries from exceeding the current latest log term.

The key element of the contract is the \text{LiveMatch} rule in \(C_3\), which demands that any change to the log must preserve all indices that are \textit{alive} in the existing log’s term. An index is alive for a given term if it is still possible for it to become committed in that term. If a quorum of replicas have all not accepted \(i\) in \(t_1\), and those replicas all vote in a greater term \(t_2\), then \(i\) becomes \textit{dead} in \(t_1\) and may be overwritten. Propose is safe because an elected leader knows that all lower-term log entries that it has not seen accepted are dead, because its quorum of voters did not accept them.
References


