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 implemention 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

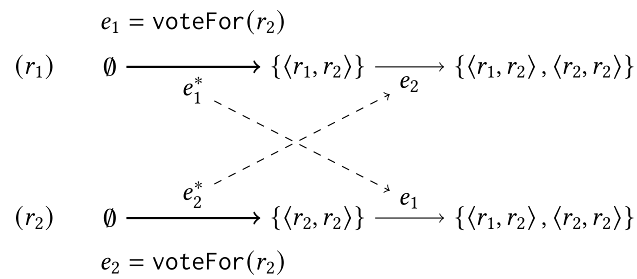

Figure 1. A leader election execution with two replicas. Replica r_1 creates event e_1 by voting for r_2—this event modifies both r_1's state (immediately) and r_2's state (after an update message is delivered). Both e_1 and e_2 are present in the traces of both r_1 and r_2, but take effect in different orders.

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.

$$
\begin{aligned}
&\mathtt{Leader}(s, r) \iff \\
&\qquad \mathtt{Quorum}(s, r) \land \neg\mathtt{DoubleQuorum}(s) \\
&\mathtt{Quorum}(s, r) \iff \\
&\qquad |\{\, r_1 \mid \langle r_1, r \rangle \in s \,\}| \geq Q \\
&\mathtt{DoubleQuorum}(s) \iff \\
&\qquad \exists r_1, r_2.\quad r_1 \neq r_2 \land \mathtt{Quorum}(s, r_1) \land \mathtt{Quorum}(s, r_2)
\end{aligned}
$$

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

$$r_1 \neq r_2 \implies \neg\mathtt{Leader}(s, r_1) \lor \neg\mathtt{Leader}(s, r_2).$$

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

$$\mathtt{Leader}(\{\langle r_1, r_2 \rangle, \langle r_2, r_2 \rangle\}, r_2).$$

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:

$$
\begin{aligned}
&\langle s_1, s_2 \rangle \in I_f \iff \\
&\qquad \forall r.\ \mathtt{Leader}(s_1, r) \implies \mathtt{Leader}(s_2, r)
\end{aligned}
$$

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-index finality.

2.3 Implementation by Replicated State Update

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

$$
\begin{aligned}
&\mathtt{voteFor}(r) := && (1)\\
&\qquad \lambda(\mathtt{self}).\ \lambda(\mathtt{state}). && (2)\\
&\qquad \textbf{assert}\ \mathtt{nonVoter}(\mathtt{self}, \mathtt{state}). && (3)\\
&\qquad \textbf{update}\ \mathtt{Insert}(\langle \mathtt{self}, r \rangle). && (4)
\end{aligned}
$$

$$
\begin{aligned}
&\mathtt{nonVoter}(r, s) &&\iff &&\forall r_1.\ \langle r, r_1 \rangle \notin s \\
&\mathtt{Insert}(v) := && &&\lambda s.\ s \cup \{v\}
\end{aligned}
$$

Once invoked, this transaction receives two runtime-provided arguments (line 2): the invoking replica's ID `self`, and its local state copy `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 `self`. More precisely, this statement performs three actions that a developer would need to implement individually in the traditional approach:

1. Insert the vote $\langle \mathtt{self}, r \rangle$ into the local state of `self`.
2. Broadcast a message to `self`'s peers, announcing that it has voted for r.

3. Run a handler on each peer that recieves the message, which adds $\langle \mathtt{self}, r\rangle$ 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 $\mathtt{Leader}(s, r)$ predicate, and thus violate I_f.

Can $\mathtt{voteFor}(r)$ create a double-vote? When r_1 executes $\mathtt{voteFor}(r)$, 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 $\mathtt{voteFor}(r)$ transaction only votes using $\mathtt{self}$. 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.

$$\begin{aligned}
&\langle s_1, s_2\rangle \in C_e(r) \iff \\
&\quad (\neg\mathtt{DoubleVote}(s_1) \implies \neg\mathtt{DoubleVote}(s_2)) \\
&\quad \wedge (\mathtt{nonVoter}(r, s_1) \implies \mathtt{nonVoter}(r, s_2))
\end{aligned}$$

$$\begin{aligned}
&\mathtt{DoubleVote}(s) \iff \\
&\quad \exists r_1, r_2, r_3. \quad r_2 \neq r_3 \wedge \langle r_1, r_2\rangle \in s \wedge \langle r_1, r_3\rangle \in s
\end{aligned}$$

As 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 $\mathtt{voteFor}(r)$.

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 $\mathtt{voteFor}(r)$, 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 $\mathtt{voteFor}(r_1)$ gives $\mathtt{nonVoter}(r_1, \mathtt{state})$.
2. $\mathtt{nonVoter}(r_1, \mathtt{state}) \wedge \langle \mathtt{state}, s_2\rangle \in C_e(r_1)$ implies that $\mathtt{nonVoter}(r_1, s_2)$.
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 $\mathtt{Leader}(s_2, r_l)$, for some candidate r_l, and $\neg\mathtt{Leader}(s_2 \cup \langle r_1, r\rangle, r_l)$. 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].

$$(r_1)\quad s_1 = \{\langle r_2, r_1\rangle\} \xrightarrow[e_1^*]{e_1 = \texttt{vote}(r_1, r_1)} \{\langle r_1, r_1\rangle, \langle r_2, r_1\rangle\}$$

$$(r_2)\quad \{\langle r_2, r_1\rangle\} \xrightarrow[e_2 = ???]{e_2^*} s_2 = \{\langle r_1, r_2\rangle, \langle r_2, r_1\rangle, \langle r_2, r_2\rangle\} \xrightarrow{e_1} s_3 = \{\langle r_1, r_1\rangle, \langle r_1, r_2\rangle, \langle r_2, r_1\rangle, \langle r_2, r_2\rangle\}$$

$$\texttt{Leader}(s_2, r_2) \quad \wedge \quad \neg\texttt{Leader}(s_3, r_2) \quad \implies \quad \langle s_2, s_3\rangle \notin I_f$$

Figure 2. This partial execution demonstrates the role of the interference contract in our proof system. The e_1 vote asserts that r_1 has not voted in s_1, but the update it creates is eventually applied to r_2's state s_2, where the assertion no-longer holds. The behavior of e_2 does not match any real transaction in our application, so we will use an interference contract that rules it out.

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

The second guarantee of ICRS, *causal consistency*, is standard for replicated state programming models [2, 4, 9]. Like identity-awareness, causal consistency can be implemented in a non-blocking manner [1] that does not interfere with *availability* in the CAP [3] sense.

We use causal consistency to ensure that replica states are correctly constrained by interference contracts. An interference contract captures the differences between states that may arise when *adding* updates. The ICRS model does not inherently guarantee eventual consistency—rather, that property that must be verified for specific applications.

3.2 Formal Execution Model

We formally model executions of ICRS applications executions as *replica trace executions*, as depicted in Fig. 1 and Fig. 2. A replica trace execution consists of a mapping of replica IDs to *traces*—sequences of update events that modify the replicas' states.

An event e in a replica's trace corresponds to either a transaction executed locally by that replica, or an update message sent by another replica's transaction. For example, in Fig. 1, the e_1 found in r_1's trace corresponds to the $\texttt{voteFor}(r_2)$ transaction executed locally by r_1, while the matching e_1 found in r_2's trace corresponds to the delivery of the update message sent by the that transaction. In either case, the event's pre- and post-states correspond to the update produced by the relevant transaction.

When events in two different traces arise from the same transaction—as with e_1 in r_1's trace and e_1 in r_2's trace—we give them the same label. We say that r_1 is e_1 *originating trace*. We assume that ICRS applications only produce replica trace executions that satisfy causal consistency.

Definition 3.1 (Causal consistency)**.** An execution X is *causally consistent* iff, for every event e_1 that preceeds an event e_2 in e_2's originating trace, it is the case that e_1 preceeds e_2 in every trace.

Eventual consistency, in contrast, is a verification goal for ICRS applications.

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 $\langle s_1, s_2\rangle \in I$.

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:

$$\{\ \langle s_1, s_2\rangle \mid \neg\texttt{DoubleVote}(s_1) \implies \neg\texttt{DoubleVote}(s_2)\ \}.$$

4 Verifiying ICRS Applications

To verify an ICRS application, the developer provides an *interference contract* that restricts replica abilities according to their IDs. For each replica ID r_1, the interference contract C defines a state-preorder $C(r_1)$. If r_1 executes a transaction using local state s_1, every remote state s_2 that the resulting update applies to is guaranteed to satisfy $\langle s_1, s_2\rangle \in C(r_1)$. At the same time, r_1 must respect the contract for any other replica r_2—the r_1 update must preserve $C(r_2)$ when applied to s_2, in case r_2 is executing a concurrent transaction of its own.

Verification proceeds by checking that every transaction T safely preserves the interference contract, according to the following rule, where S is the set of all application states and

D is the set of all pairs of distinct replica IDs:

$$\frac{\begin{array}{c}\forall s_1, s_2 \in S. \quad \forall \langle r_1, r_2 \rangle \in D. \\ (T(r_1, s_1) \rightsquigarrow u \wedge \langle s_1, s_2 \rangle \in C(r_1)) \\ \Longrightarrow \\ \langle s_2, u(s_2) \rangle \in (C(r_2) \cap I)\end{array}}{C \vdash \mathtt{Safe}(T, I)}$$

The notation $T(r_1, s_1) \rightsquigarrow u$ 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 $u \in S \longrightarrow S$. 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 $C \vdash \mathtt{Safe}(T_1, I) \wedge \cdots \wedge C \vdash \mathtt{Safe}(T_n, I)$, 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 proceedure 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:

- `votes` $\in \mathcal{P}(\mathtt{Rid} \times \mathtt{Rid} \times \mathtt{Nat})$ The set of votes. An element $\langle r_v, r_c, t \rangle$ represents r_v 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 in t is present.
- `term` $\in$ `Nat` The latest term in which the log has been modified.
- `accepts` $\in \mathcal{P}(\mathtt{Rid} \times \mathtt{Nat} \times \mathtt{Nat})$ A record of what prefix of the log has been *accepted* (i.e. witnessed) by each replica in each term. The entry $\langle r, t, i \rangle$ means that r has accepted the i-length prefix of the log written in term t.
- `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. $\mathtt{Leader}(s, r, t)$ states that a given replica ID is the (only) elected leader for a term t—the definition is similar to that of $\mathtt{Leader}(s, r)$ from Sec. 2. `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.

$$\begin{array}{l}\mathtt{Committed}(s \in \mathtt{State}, t \in \mathtt{Nat}, i \in \mathtt{Nat}) \iff \\ \quad \exists t_1 \in \mathtt{Nat}. \quad t_1 \leq t \\ \quad \wedge \mid \{ \, r \mid \exists i_1. \; i \leq i_1 \wedge \langle r, t_1, i_1 \rangle \in s.\mathtt{accepts} \, \} \mid \geq Q\end{array}$$

5.2 Distributed Log Consensus

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

$$\begin{array}{l}\langle s_1, s_2 \rangle \in I_c \iff \\ \quad \forall i \in \mathtt{Nat}. \, \forall t \in \mathtt{Nat}. \\ \quad \mathtt{Committed}(s_1.\mathtt{accepts}, t, i) \\ \quad\quad \Longrightarrow \mathtt{PrefixMatch}(i, s_1.\mathtt{log}, s_2.\mathtt{log})\end{array}$$

Like our example in Sec. 2, the I_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 $\mathtt{vote}(r, t)$ transaction is analogous to the $\mathtt{voteFor}(r)$ transaction in Sec. 2, but in this case leaders are elected for particular terms.

Elected leaders use the $\mathtt{propose}(l, t)$ 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, $\mathtt{propose}(l, t)$ issues the update $\mathtt{NewLog}(l, t)$, 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 `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 $\mathtt{accept}(i)$ transaction is used a 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

$\mathtt{vote}(r, t) :=$
  $\lambda(\mathtt{self}).\ \lambda(\mathtt{state}).$
  assert $\mathtt{nonVoter}(\mathtt{state}, \mathtt{self}, t).$
  update $\mathtt{votes.Insert}(\langle \mathtt{self}, r, t\rangle)$

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

$\mathtt{accept}(i) :=$
  $\lambda(\mathtt{self}).\ \lambda(\mathtt{state}).$
  assert $\mathtt{nonVoterOver}(\mathtt{state}, \mathtt{self}, \mathtt{state.term})$
  assert $\mathtt{HasSize}(\mathtt{state.log}, i).$
  update $\mathtt{accepts.Insert}(\langle \mathtt{self}, \mathtt{state.term}, i\rangle)$

$\mathtt{NewLog}(l, t) := \quad \lambda s.$
  $\mathtt{case}\ (t \geq s.\mathtt{term}) \to s\{\mathtt{term} = t, \mathtt{log} = l\}$
  $\mathtt{case}\ (t < s.\mathtt{term}) \to s$

Figure 3. Transactions and updates for the Raft implementation.

$C_1(r, s_1, s_2) \iff \quad (\textit{Vote Safety})$
  $\mathtt{VoteFreeze}(r, s_1.\mathtt{votes}, s_2.\mathtt{votes})$
  $\neg\mathtt{DoubleVote}(s_1.\mathtt{votes}) \implies \neg\mathtt{DoubleVote}(s_2.\mathtt{votes})$
  $\wedge\ s_1.\mathtt{votes} \subseteq s_2.\mathtt{votes}$

$C_2(r, s_1, s_2) \iff \quad (\textit{Accept Safety})$
  $\mathtt{AcceptFreeze}(r, s_1.\mathtt{accepts}, s_2.\mathtt{accepts})$
  $\wedge\ \mathtt{VoterRestrict}(s_1.\mathtt{votes}, s_1.\mathtt{accepts}, s_2.\mathtt{accepts})$
  $\wedge\ s_1.\mathtt{accepts} \subseteq s_2.\mathtt{accepts}$
  $\wedge\ \mathtt{AcceptTerm}(s_1.\mathtt{accepts}, s_1.\mathtt{term})$
    $\implies \mathtt{AcceptTerm}(s_2.\mathtt{accepts}, s_2.\mathtt{term})$

$C_3(r, s_1, s_2) \iff \quad (\textit{Log Safety})$
  $\mathtt{LiveMatch}(s_2.\mathtt{votes}, s_2.\mathtt{accepts}, s_1.\mathtt{term}, s_1.\mathtt{log}, s_2.\mathtt{log})$
  $\wedge\ s_1.\mathtt{term} \leq s_2.\mathtt{term}$
  $\wedge\ \mathtt{LiveBound}(s_1.\mathtt{votes}, s_1.\mathtt{accepts}, s_1.\mathtt{log})$
    $\implies \mathtt{LiveBound}(s_2.\mathtt{votes}, s_2.\mathtt{accepts}, s_2.\mathtt{log})$

$C_4(r, s_1, s_2) \iff \quad (\textit{Leader Log})$
  $\mathtt{Leader}(s_1, r, s_2.\mathtt{term})$
  $\implies s_1.\mathtt{term} = s_2.\mathtt{term} \wedge s_1.\mathtt{log} = s_2.\mathtt{log}$

Figure 4. Interference contract for verifying that the Raft implementation satisfies I_c as a replica trace invariant.

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

5.4 Verifying Eventual Consistency

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

$$s \in G(\mathtt{NewLog}(l, t)) \iff s.\mathtt{term} \neq t \vee \mathtt{logPrefix}(s.\mathtt{log}, l)$$

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 $\mathtt{NewLog}(l_1, t)$ and $\mathtt{NewLog}(l_2, t)$, where $l_1 \neq l_2$.

5.5 Verifying Distributed Log Consensus

The interference contract for verifying that Raft satisfies the I_c trace invariant is shown in Fig 4.

The $\mathtt{VoteFreeze}(r, v_1, v_2)$ and $\mathtt{AcceptFreeze}(r, m_1, m_2)$ rules express that remote replicas will not vote or accept, respectively, using r's ID. The $\mathtt{VoterRestrict}(r, m_1, m_2)$ 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 nonVoterOver assertion in $\mathtt{accept}(i)$). AcceptTerm forbids entries from exceeding the current latest log term.

The key element of the contract is the LiveMatch rule in C_3, which demands that any change to the log must preserve all indices that are *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 *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

[1] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.

[2] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 6:1–6:16, New York, NY, USA, 2015. ACM.

[3] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[4] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. '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*, POPL '16, pages 371–384, New York, NY, USA, 2016. ACM.

[5] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 1–17, New York, NY, USA, 2015. ACM.

[6] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, may 1998.

[7] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[8] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA, 2014. USENIX Association.

[9] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 413–424, New York, NY, USA, 2015. ACM.

[10] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. 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*, PLDI 2018, page 662–677, New York, NY, USA, 2018. Association for Computing Machinery.

[11] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend synchrony: Synchronous verification of asynchronous distributed programs. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[12] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 357–368, New York, NY, USA, 2015. Association for Computing Machinery.

[13] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 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*, CPP 2016, page 154–165, New York, NY, USA, 2016. Association for Computing Machinery.