15-812 Term Paper: Specifying and proving cluster membership for the Raft distributed consensus algorithm

Brandon Amos and Huanchen Zhang*

2015/05/10

*Equal contribution, alphabetical order.

Contents

1 Introduction

Distributed consensus is popular in today's world as many large-scale production systems rely on reaching consensus among a set of decentralized servers. Consensus algorithms are notoriously difficult to correctly implement and formal verification methods are helpful in proving properties of the algorithms.

Raft is a newly released consensus algorithm that is beginning to be adopted in large-scale systems [OO14]. A partial formal specification of Raft is presented in [Ong14] and used in hand proofs for a subset of properties.

In this report, we add new functionality to the formal specification in §3. We prove (by hand) a safety property of there being at most one leader per term under our modifications in §4.1. We describe a proof sketch in §4.2 showing that at any point, a leader can be elected in the future.

2 Background

2.1 The Raft Consensus Algorithm

Raft is consensus algorithm that allows a collection of machines to work as a coherent group that can survive failures of some members and is presented at USENIX ATC'14 [OO14] and further expanded on in Diego Ongaro's thesis [Ong14]. The Secret Lives of Data [sec] provides a visual walkthrough and introduction to Raft.

Raft has moved beyond academia and is being implemented and deployed in large-scale production systems, as described on the website [raf].

Some important concepts and terms for understanding Raft are:

- **Replicated Log.** Each node maintains a log that contains values and configuration entries. Because the system is distributed, the logs aren't guaranteed to be consistent on every server. Log entries can be **committed**, which means that a majority of the nodes agree on the entry. A majority of nodes is also called a **quorum**.
- **Server states.** Servers in the cluster exist in the following three states.
  - **Leader.** The leader receives requests from external entities to append values to the replicated log.
  - **Follower.** Followers receive commands from the leader to add new entries to their logs.
  - **Candidate.** If a follower doesn't hear from a leader within a specified interval, it times out and becomes a candidate.
- **Configuration.** The configuration is the set of servers in the Raft system. The protocol allows servers to be added and removed from the system.

In this report, we study adding and removing servers from the cluster. Adding and removing servers is done by operating on one server at a time and keeping track of the configuration with the normal log replication mechanisms. The RPC's for adding and removing servers are fully described in Figure 1.

AddServer RPC

Invoked by admin to add a server to the cluster configuration.

Arguments:

newServer	address of server to add to configuration

Results:

status	OK if server was added successfully
leaderHint	address of recent leader, if known

Receiver implementation:

1. Reply NOT_LEADER if not leader (§6.2)
2. Catch up new server for fixed number of rounds. Reply TIMEOUT if new server does not make progress for an election timeout or if the last round takes longer than the election timeout. (§4.2.1)
3. Wait until previous configuration in log is committed (§4.1)
4. Append new configuration entry to log (old configuration plus newServer), commit it using majority of new configuration (§4.1)
5. Reply OK

RemoveServer RPC

Invoked by admin to remove a server from the cluster configuration.

Arguments:

oldServer	address of server to remove from configuration

Results:

status	OK if server was removed successfully
leaderHint	address of recent leader, if known

Receiver implementation:

1. Reply NOT_LEADER if not leader (§6.2)
2. Wait until previous configuration in log is committed (§4.1)
3. Append new configuration entry to log (old configuration without oldServer), commit it using majority of new configuration (§4.1)
4. Reply OK and, if this server was removed, step down (§4.2.2)

Figure 1: Implementation of Raft's Add and Remove RPC's. Copied from Figure 4.1 of [Ong14] and included here for completeness.

2.1.1 Safety and Availability

Safety and availability (or liveness) are fundamental properties systems that are important to formally verify [AS87]. The safety property of Raft we focus on is that two leaders can never be elected in the same term. An availability property of Raft is that a leader can be elected at some point in the future.

2.2 Temporal Logic of Actions (TLA)

Lamport's temporal logic of actions (TLA) [Lam94] is a logic for specifying and reasoning about concurrent systems. Figure 2 summarizes minimal syntax and semantics of TLA. TLA+ is formal specification language that describes system behavior using TLA [Lam02]. TLA+ breaks distributed algorithms into state transition functions that specify all possible behaviors of the system. The TLA+ Model Checker (TLC) [YML99] exhaustively checks whether a property or invariant holds. The TLA+ Proof System [CDLM08] mechanically checks TLA+ proofs. [Lam00] provides a helpful summary and description for reading and writing TLA+.

Appendix B of Ongaro's thesis [Ong14] provides a TLA+ specification and hand-written proofs of a subset of Raft's properties and features. §8 of the thesis provides informal arguments about correctness. For completeness (and convenience), this report includes the original TLA+ specification in Appendix A.

Syntax

$\langle formula \rangle \triangleq \langle predicate \rangle \mid \Box[\langle action \rangle]_{\langle state\ function \rangle} \mid \neg \langle formula \rangle$
$\quad \mid \langle formula \rangle \wedge \langle formula \rangle \mid \Box \langle formula \rangle$

$\langle action \rangle \triangleq$ boolean-valued expression containing constant symbols, variables, and primed variables

$\langle predicate \rangle \triangleq \langle action \rangle$ with no primed variables $\mid$ $Enabled\ \langle action \rangle$

$\langle state\ function \rangle \triangleq$ nonboolean expression containing constant symbols and variables

Semantics

$s[\![f]\!] \triangleq f(\forall\, `v\text{'} : s[\![v]\!]/v)$

$s[\![\mathcal{A}]\!]t \triangleq \mathcal{A}(\forall\, `v\text{'} : s[\![v]\!]/v,\ t[\![v]\!]/v')$

$\models \mathcal{A} \triangleq \forall s, t \in \mathbf{St} : s[\![\mathcal{A}]\!]t$

$\sigma[\![F \wedge G]\!] \triangleq \sigma[\![F]\!] \wedge \sigma[\![G]\!]$

$\sigma[\![\neg F]\!] \triangleq \neg \sigma[\![F]\!]$

$\models F \triangleq \forall \sigma \in \mathbf{St}^{\infty} : \sigma[\![F]\!]$

$s[\![Enabled\ \mathcal{A}]\!] \triangleq \exists t \in \mathbf{St} : s[\![\mathcal{A}]\!]t$

$\langle s_0, s_1, \ldots \rangle[\![\Box F]\!] \triangleq \forall n \in \mathsf{Nat} : \langle s_n, s_{n+1}, \ldots \rangle[\![F]\!]$

$\langle s_0, s_1, \ldots \rangle[\![\mathcal{A}]\!] \triangleq s_0[\![\mathcal{A}]\!]s_1$

Additional notation

$p' \triangleq p(\forall\, `v\text{'} : v'/v)$

$[\mathcal{A}]_f \triangleq \mathcal{A} \vee (f' = f)$

$\langle \mathcal{A} \rangle_f \triangleq \mathcal{A} \wedge (f' \neq f)$

$Unchanged\ f \triangleq f' = f$

$\Diamond F \triangleq \neg \Box \neg F$

$F \leadsto G \triangleq \Box(F \Rightarrow \Diamond G)$

$\mathrm{WF}_f(\mathcal{A}) \triangleq \Box\Diamond\langle \mathcal{A} \rangle_f \vee \Box\Diamond\neg Enabled\ \langle \mathcal{A} \rangle_f$

$\mathrm{SF}_f(\mathcal{A}) \triangleq \Box\Diamond\langle \mathcal{A} \rangle_f \vee \Diamond\Box\neg Enabled\ \langle \mathcal{A} \rangle_f$

where f is a $\langle state\ function \rangle$
$\mathcal{A}$ is an $\langle action \rangle$
F and G are $\langle formula \rangle$s
p is a $\langle state\ function \rangle$ or $\langle predicate \rangle$
$s, s_0, s_1, \ldots$ are states
σ is a behavior
$(\forall\, `v\text{'} : \ldots/v,\ \ldots/v')$ denotes substitution for all variables v

Figure 2: Summary of TLA's simple syntax and semantics. Copied from Figure 4 of [Lam94] and included here for completeness.

3 Adding cluster membership changes to Raft's formal specification

We have extended Raft's formal TLA+ specification to allow server configuration changes. For completeness, Appendix B provides our modified specification.

3.1 Modeling Network Messages

We utilize the existing specification for messaging between Raft nodes by using the `Send`, `Discard`, and `Reply` helper functions. Messages in the system are represented as a bag in `messages` that maps a message's content to an integer. This integer counts the number of active messages in the system and is initialized to one, incremented by one when a message is duplicated or sent again, and decremented by one when a message is discarded or replied to.

Network packets can be duplicated or dropped, which the TLA+ specification models with `Duplicate` and `Drop` in the state transition function.

3.2 New Variables

We have added the following new variables and constants to the specification. Our original modifications included other variables that introduced a new state for detached servers and kept track of additional indexes. However, we realized these could be deduced mathematically from other variables in the system.

- `NumRounds`. The number of rounds to catch each server up by.
- `InitServer` and `Server`. Previously, there was only a single constant describing the set of servers in the system. We have modified this to describe both an initial and global set of servers that can be added and removed.
- `ValueEntry` and `ConfigEntry`. Previously, the log only contained homogeneous entries. Now, configuration changes are also stored in the log and each entry is now identified as either a value or config with a `type` metadata.
- `CatchupRequest`, `CatchupResponse`, and `CheckOldConfig`. New message types in the system to catch up servers and check if the old config have been committed.

3.3 Initial state of the system

We have only slightly modified the system initialization in `Init` to correctly handle the changed set of servers. Every variable is initialized to contain information for the global set of servers, even if they aren't in the initial configuration, so that the lists do not have to be resized every time a server receives a configuration change. This prevents some corner cases when server receives a configuration change in it's log that doesn't get committed that is then overwritten by another log entry.

3.4 State Transitions

In the `Next` state transition definitions, we modify the existential operators to operate on the global set of servers. Some servers might not be in any configurations, so we add restrictions to the state transition functions.

- `Timeout`. A server can only timeout, become a candidate, and start a new election if it is in its own configuration.
- `RequestVote`. Candidates only request votes from servers in their configuration.
- `AppendEntries`. Leaders only send new log entries to servers in their configuration.
- `BecomeLeader`. A candidate can only become a leader if they receive votes from a majority of their quorum.

- `ClientRequest`. Unmodified, only leaders receive requests from clients to add new values to the replicated state machine.

- `AdvanceCommitIndex`. Leaders can advance the commit index if all servers in their config agree.

3.4.1 AddNewServer

We have added a new state transition function to add a new server to the system. This can be called when some server i is the leader and adds a new server that's not in it's configuration. This sends a `CatchupRequest` message to the server to be added with log entries to append.

The first time this is called, `nextIndex[i][j]` will be 0 and the entire committed log will be sent. However, this can be called multiple times before a server is added when i is still a leader, since j will not be added to it's configuration until the server is sufficiently caught up. Therefore, the leader uses `nextIndex[i][j]` to keep track of the new server's state so that duplicate requests are not harmful.

Leader i adds a new server j to the cluster.

$$
\begin{aligned}
& \mathit{AddNewServer}(i, j) \triangleq \\
& \quad \land \mathit{state}[i] = \mathit{Leader} \\
& \quad \land j \notin \mathit{GetConfig}(i) \\
& \quad \land \mathit{currentTerm}' = [\mathit{currentTerm} \text{ EXCEPT } ![j] = 1] \\
& \quad \land \mathit{votedFor}' = [\mathit{votedFor} \text{ EXCEPT } ![j] = \mathit{Nil}] \\
& \quad \land \mathit{Send}([\mathit{mtype} \mapsto \mathit{CatchupRequest}, \\
& \qquad\qquad \mathit{mterm} \mapsto \mathit{currentTerm}[i], \\
& \qquad\qquad \mathit{mlogLen} \mapsto \mathit{matchIndex}[i][j], \\
& \qquad\qquad \mathit{mentries} \mapsto \mathit{SubSeq}(\mathit{log}[i], \mathit{nextIndex}[i][j], \mathit{commitIndex}[i]), \\
& \qquad\qquad \mathit{mcommitIndex} \mapsto \mathit{commitIndex}[i], \\
& \qquad\qquad \mathit{msource} \mapsto i, \\
& \qquad\qquad \mathit{mdest} \mapsto j, \\
& \qquad\qquad \mathit{mrounds} \mapsto \mathit{NumRounds}]) \\
& \quad \land \text{UNCHANGED } \langle \mathit{state}, \mathit{leaderVars}, \mathit{logVars}, \mathit{candidateVars} \rangle
\end{aligned}
$$

3.4.2 DeleteServer

Deleting a server is simpler than adding a server because no catching up needs to be done. The system needs to wait until a previous configuration change has been committed. One edge case that we haven't specified is when a leader is asked to delete itself.

Leader i removes a server j (possibly itself) from the cluster.

$$
\begin{aligned}
& \mathit{DeleteServer}(i, j) \triangleq \\
& \quad \land \mathit{state}[i] = \mathit{Leader} \\
& \quad \land \mathit{state}[j] \in \{\mathit{Follower}, \mathit{Candidate}\} \\
& \quad \land j \in \mathit{GetConfig}(i)
\end{aligned}
$$

$\land j \neq i$ *TODO*: A leader cannot remove itself.

$$
\begin{aligned}
&\land\ Send([mtype \mapsto CheckOldConfig,\\
&\qquad mterm \mapsto currentTerm[i],\\
&\qquad madd \mapsto \text{FALSE},\\
&\qquad mserver \mapsto j,\\
&\qquad msource \mapsto i,\\
&\qquad mdest \mapsto i])\\
&\land\ \text{UNCHANGED}\ \langle serverVars, candidateVars, leaderVars, logVars \rangle
\end{aligned}
$$

3.5 Modifying helper functions

3.5.1 Quorum

With static configurations, the quorum remains constant throughout execution. However, with dynamically changing configurations, a quorum is specific to each server's current view of the system, so we have added a parameter to the `Quorum` helper function definition so each server can compute a quorum for it's current configuration.

The set of all quorums for a server configuration.
This just calculates simple majorities, but the only
important property is that every quorum overlaps with every other.

$$Quorum(config) \triangleq \{i \in \text{SUBSET}\ (config) : Cardinality(i) * 2 > Cardinality(config)\}$$

3.5.2 Getting a server's configuration

Servers immediately start using configuration entries as they are appended to their logs, before they're committed. If a server's log has no configuration entries, the initial set of servers is used. We introduce the following helper functions `GetMaxConfigIndex` and `GetConfig` because many portions of the handlers and state transition functions require the server's configuration.

Return the index of the latest configuration in server i's *log*.

$$
\begin{aligned}
&GetMaxConfigIndex(i) \triangleq\\
&\quad \text{LET}\ configIndexes \triangleq \{index \in 1\,..\,Len(log[i]) : log[i][index].type = ConfigEntry\}\\
&\quad \text{IN}\quad \text{IF}\ configIndexes = \{\}\ \text{THEN}\ 0\\
&\qquad\qquad \text{ELSE}\ \ Max(configIndexes)
\end{aligned}
$$

Return the configuration of teh latest configuration in server i's *log*.

$$
\begin{aligned}
&GetConfig(i) \triangleq\\
&\ \text{IF}\ GetMaxConfigIndex(i) = 0\ \text{THEN}\ InitServer\\
&\quad \text{ELSE}\ \ log[i][GetMaxConfigIndex(i)].value
\end{aligned}
$$

3.6 Handlers for configuration changes

We have introduced the following handlers for the new messages in the system.

3.6.1 Handling `CatchupRequest` messages

When a detached server receives this message, it should first check if the message is still valid, by checking `mterm` in the message. If this agrees, the server will appropriately overwrite and/or append the new entries (`mentries`) to it's log and respond to the leader indicating the current log position and that it has one less round to complete.

Detached server i receives a *CatchupRequest* from leader j.

$$
\begin{aligned}
& \mathit{HandleCatchupRequest}(i, j, m) \triangleq \\
& \quad \lor \land\ m.\mathit{mterm} < \mathit{currentTerm}[i] \\
& \qquad \land\ \mathit{Reply}([\mathit{mtype} \mapsto \mathit{CatchupResponse}, \\
& \qquad\qquad\quad \mathit{mterm} \mapsto \mathit{currentTerm}[i], \\
& \qquad\qquad\quad \mathit{msuccess} \mapsto \text{FALSE}, \\
& \qquad\qquad\quad \mathit{mmatchIndex} \mapsto 0, \\
& \qquad\qquad\quad \mathit{msource} \mapsto i, \\
& \qquad\qquad\quad \mathit{mdest} \mapsto j, \\
& \qquad\qquad\quad \mathit{mroundsLeft} \mapsto 0], \\
& \qquad\qquad\quad m) \\
& \qquad \land\ \text{UNCHANGED}\ \langle \mathit{serverVars}, \mathit{candidateVars}, \\
& \qquad\quad \mathit{leaderVars}, \mathit{logVars} \rangle \\
& \quad \lor \land\ m.\mathit{mterm} \geq \mathit{currentTerm}[i] \\
& \qquad \land\ \mathit{currentTerm}' = [\mathit{currentTerm}\ \text{EXCEPT}\ ![i] = m.\mathit{mterm}] \\
& \qquad \land\ \mathit{log}' = [\mathit{log}\ \text{EXCEPT}\ ![i] = \mathit{SubSeq}(\mathit{log}[i], 1, m.\mathit{mlogLen}) \circ m.\mathit{mentries}] \\
& \qquad \land\ \mathit{Reply}([\mathit{mtype} \mapsto \mathit{CatchupResponse}, \\
& \qquad\qquad\quad \mathit{mterm} \mapsto \mathit{currentTerm}[i], \\
& \qquad\qquad\quad \mathit{msuccess} \mapsto \text{TRUE}, \\
& \qquad\qquad\quad \mathit{mmatchIndex} \mapsto \mathit{Len}(\mathit{log}[i]), \\
& \qquad\qquad\quad \mathit{msource} \mapsto i, \\
& \qquad\qquad\quad \mathit{mdest} \mapsto j, \\
& \qquad\qquad\quad \mathit{mroundsLeft} \mapsto m.\mathit{mrounds} - 1], \\
& \qquad\qquad\quad m) \\
& \qquad \land\ \text{UNCHANGED}\ \langle \mathit{state}, \mathit{votedFor}, \mathit{candidateVars}, \mathit{leaderVars}, \\
& \qquad\quad \mathit{commitIndex} \rangle
\end{aligned}
$$

3.6.2 Handling `CatchupResponse` messages

When a leader receives the `CatchupResponse` message, it checks if the server indicated it was successful in `msuccess`, then makes sure the `mmatchIndex` is correctly set. If so, it will send another request to the server with new log entries to catch up if there are still rounds remaining. Otherwise, it will send a message to itself to wait until any uncommitted entries in it's log have been committed.

Leader i receives a *CatchupResponse* from detached server j.

$HandleCatchupResponse(i, j, m) \triangleq$

A real system checks for progress every timeout interval.
Assume that if this response is called, the new server
has made progress.

$\land \lor \land m.msuccess$

$\quad\quad \land \lor \land m.mmatchIndex \neq commitIndex[i]$

$\quad\quad\quad\quad \land m.mmatchIndex \neq matchIndex[i][j]$

$\quad\quad\quad \lor m.mmatchIndex = commitIndex[i]$

$\quad\quad \land state[i] = Leader$

$\quad\quad \land m.mterm = currentTerm[i]$

$\quad\quad \land j \notin GetConfig(i)$

$\quad\quad \land nextIndex' = [nextIndex \text{ EXCEPT } ![i][j] = m.mmatchIndex + 1]$

$\quad\quad \land matchIndex' = [matchIndex \text{ EXCEPT } ![i][j] = m.mmatchIndex]$

$\quad\quad \land \lor \land m.mroundsLeft \neq 0$

$\quad\quad\quad\quad \land Reply([mtype \mapsto CatchupRequest,$

$\quad\quad\quad\quad\quad\quad mterm \mapsto currentTerm[i],$

$\quad\quad\quad\quad\quad\quad mentries \mapsto SubSeq(log[i], nextIndex[i][j], commitIndex[i]),$

$\quad\quad\quad\quad\quad\quad mLogLen \mapsto nextIndex[i][j] - 1,$

$\quad\quad\quad\quad\quad\quad msource \mapsto i,$

$\quad\quad\quad\quad\quad\quad mdest \mapsto j,$

$\quad\quad\quad\quad\quad\quad mrounds \mapsto m.mroundsLeft],$

$\quad\quad\quad\quad\quad\quad m)$

$\quad\quad\quad \lor \land m.mroundsLeft = 0$

A real system makes sure the final call to this handler is
received after a timeout interval.
We assume that if a timeout happened, the message
has already been dropped.

$\quad\quad\quad\quad \land Reply([mtype \mapsto CheckOldConfig,$

$\quad\quad\quad\quad\quad\quad mterm \mapsto currentTerm[i],$

$\quad\quad\quad\quad\quad\quad madd \mapsto \text{TRUE},$

$\quad\quad\quad\quad\quad\quad mserver \mapsto j,$

$\quad\quad\quad\quad\quad\quad msource \mapsto i,$

$\quad\quad\quad\quad\quad\quad mdest \mapsto i], m)$

$\quad\quad \land \text{UNCHANGED } \langle elections \rangle$

$\quad \lor \land \lor \neg m.msuccess$

$\quad\quad\quad \lor \land \lor m.mmatchIndex = commitIndex[i]$

$\quad\quad\quad\quad\quad \lor m.mmatchIndex = matchIndex[i][j]$

$\quad\quad\quad\quad \land m.mmatchIndex \neq commitIndex[i]$

$\quad\quad\quad \lor state[i] \neq Leader$

$$
\begin{array}{l}
\quad\quad\quad \lor m.mterm \neq currentTerm[i] \\
\quad\quad\quad \lor j \in GetConfig(i) \\
\quad\quad \land Discard(m) \\
\quad\quad \land \text{UNCHANGED } \langle leaderVars \rangle \\
\quad \land \text{UNCHANGED } \langle serverVars, candidateVars, logVars \rangle
\end{array}
$$

3.6.3 Handling `CheckOldConfig` messages

This handler causes the leader to wait until an uncommitted configuration is committed before adding a new entry. This is used both for adding and removing servers. If there is still an uncommitted entry, the leader will send itself another message to check again in the future. In a real system, this could be implemented by using a background thread on the server that sleeps and periodically checks, but this is nontrivial to model in the TLA+ spec and is equivalent to sending itself a message, even though the message can be duplicated or dropped.

Leader i receives a *CheckOldConfig* message.

$$
\begin{array}{l}
HandleCheckOldConfig(i, m) \triangleq \\
\quad \lor \land state[i] \neq Leader \lor m.mterm = currentTerm[i] \\
\quad\quad \land Discard(m) \\
\quad\quad \land \text{UNCHANGED } \langle serverVars, candidateVars, leaderVars, logVars \rangle \\
\quad \lor \land state[i] = Leader \land m.mterm = currentTerm[i] \\
\quad\quad \land \lor \land GetMaxConfigIndex(i) \leq commitIndex[i] \\
\quad\quad\quad\quad \land \text{LET } newConfig \triangleq \text{IF } m.madd \text{ THEN UNION } \{GetConfig(i), \{m.mserver\}\} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{ELSE } GetConfig(i) \setminus \{m.mserver\} \\
\quad\quad\quad\quad\quad\quad newEntry \triangleq [term \mapsto currentTerm[i], type \mapsto ConfigEntry, value \mapsto newConfig] \\
\quad\quad\quad\quad\quad\quad newLog \triangleq Append(log[i], newEntry) \\
\quad\quad\quad\quad\quad \text{IN } log' = [log \text{ EXCEPT } ![i] = newLog] \\
\quad\quad\quad\quad \land Discard(m) \\
\quad\quad\quad\quad \land \text{UNCHANGED } \langle commitIndex \rangle \\
\quad\quad\quad \lor \land GetMaxConfigIndex(i) > commitIndex[i] \\
\quad\quad\quad\quad \land Reply([mtype \mapsto CheckOldConfig, \\
\quad\quad\quad\quad\quad\quad\quad mterm \mapsto currentTerm[i], \\
\quad\quad\quad\quad\quad\quad\quad madd \mapsto m.madd, \\
\quad\quad\quad\quad\quad\quad\quad mserver \mapsto m.mserver, \\
\quad\quad\quad\quad\quad\quad\quad msource \mapsto i, \\
\quad\quad\quad\quad\quad\quad\quad mdest \mapsto i], \\
\quad\quad\quad\quad\quad\quad\quad m) \\
\quad\quad\quad\quad \land \text{UNCHANGED } \langle logVars \rangle \\
\quad\quad \land \text{UNCHANGED } \langle serverVars, candidateVars, leaderVars \rangle
\end{array}
$$

3.7 Mitigating effects of disruptive servers

Configuration changes can servers that have been removed to cause suboptimal (but still correct) system performance, as illustrated in Figure 3.

By studying our new specification, we have added a slight modification to the Raft algorithm to lessen the impacts disruptive servers can have: Servers can only timeout if they are in their own configuration.

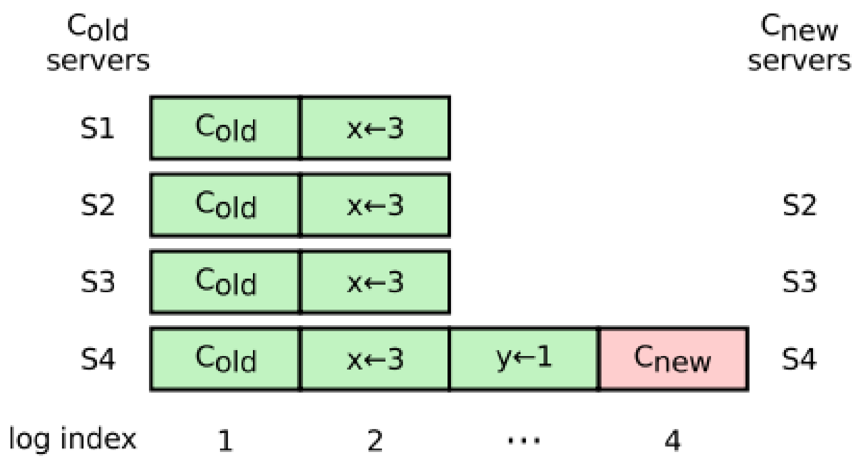

Figure 3: An example of how a server can be disruptive even before the C_{new} log entry has been committed. The figure shows the removal of S1 from a four-server cluster. S4 is leader of the new cluster and has created the C_{new} entry in its log, but it hasn't yet replicated that entry. Even before C_{new} is committed, S1 can time out, increment its term, and send this larger term number to the new cluster, forcing S4 to step down. **Figure and description copied from Figure 4.7 of [Ong14] and included here for completeness.**

3.8 Model checking the specification

We have used the TLC model checker to validate simple cases of our modified specification. We created invariants that we knew would be broken so that we could obtain a traceback of the operations and messages that caused the point to be reached. One example is that a server that's not in the initial configuration eventually receives log entries because it has been added to the cluster.

4 Proofs

4.1 Safety: There is never more than one leader.

Lemma 1. *Let* $n \geq 2$, $c_1 = \{1, \ldots, n\}$, $c_2 = \{1, \ldots, n-1\}$. *If* $s \in \mathit{Quorum}(c_1)$, $t \in \mathit{Quorum}(c_2)$, *then* $s \cap t \neq \emptyset$.

Proof.

$$|s| \geq \left\lfloor \frac{n}{2} \right\rfloor + 1$$

$$|t| \geq \left\lfloor \frac{n-1}{2} \right\rfloor + 1$$

$$|s| + |t| \geq n + 1$$

Since there are only n unique elements in $c_1 \cup c_2$, $s \cap t \neq \emptyset$. □

Lemma 2. *Let* $n \geq 1$, $c_1 = \{1, \ldots, n\}$, $c_2 = \{1, \ldots, n+1\}$. *If* $s \in Quorum(c_1)$, $t \in Quorum(c_2)$, *then* $s \cap t \neq \emptyset$.

Proof.

$$|s| \geq \left\lfloor \frac{n}{2} \right\rfloor + 1$$

$$|t| \geq \left\lfloor \frac{n+1}{2} \right\rfloor + 1$$

$$|s| + |t| \geq n + 2$$

Since there are only $n+1$ unique elements in $c_1 \cup c_2$, $s \cap t \neq \emptyset$. □

Lemma 3. *Let* $n \geq 1$, $c_1 = \{1, \ldots, n-1\}$, $c_2 = \{1, \ldots, n+1\}$. *If* $s \in Quorum(c_1)$, $t \in Quorum(c_2)$, *then* $s \cap t \neq \emptyset$.

Proof.

$$|s| \geq \left\lfloor \frac{n-1}{2} \right\rfloor + 1$$

$$|t| \geq \left\lfloor \frac{n+1}{2} \right\rfloor + 1$$

$$|s| + |t| \geq n + 2$$

Since there are only $n+1$ unique elements in $c_1 \cup c_2$, $s \cap t \neq \emptyset$. □

Lemma 4. *A quorum cannot be formed based on a stale config (i.e. a config that is before the latest committed config)*

Proof. Let C_{latest} be the latest committed config and $C_{\text{latest}-1}$ be the config that is committed right before C_{latest}.

Suppose $C_{\text{latest}} = \{1, \ldots, n\}$. Then, $C_{\text{latest}-1}$ can either be $\{1, \ldots, n-1\}$ or $\{1, \ldots, n+1\}$. For simplicity, assume the last server is the one that changes.

Since C_{latest} is committed, at least $\lfloor n/2 \rfloor + 1$ servers have C_{latest} in their logs.

- **Case 1.** $C_{\text{latest}-1} = \{1, \ldots, n-1\}$. In order to form a quorum based on $C_{\text{latest}-1}$, it requires at least $\left\lfloor \frac{n-1}{2} \right\rfloor + 1$ votes.

  However, any server with C_{latest} in its log won't vote yes because of the "Election Restriction" (§3.6.1 in [Ong14]) that "the voter denies its vote if its own log is more up-to-date than that of the candidate."

Therefore, it can only get at most $n - \lfloor n/2 \rfloor - 1$ votes.

Since

$$\left(n - \left\lfloor \frac{n}{2} \right\rfloor - 1\right) - \left(\left\lfloor \frac{n-1}{2} \right\rfloor - 1\right) = -1 < 0,$$

it can never get enough votes to form a quorum based on $C_{\text{latest}-1}$.

- **Case 2.** $C_{\text{latest}-1} = \{1, \ldots, n+1\}$. Similar argument as in **Case 1**.

Therefore, as long as C_{latest} is committed, a quorum cannot be formed based on $C_{\text{latest}-1}$. Induction can show that any config prior to C_{latest} cannot be the basis to form a quorum. □

Lemma 5. *Let C_{latest} be the latest committed config. Let C_{new} be any uncommitted config in the system, suppose $C_{\text{latest}} = \{1, \ldots, n\}$. Then, C_{new} is either $\{1, \ldots, n-1\}$ or $\{1, \ldots, n+1\}$. For simplicity, assume the last server is the one that changes.*

Proof. By Lemma 4, since any stale config cannot be the basis of a quorum, any leader before a newer config gets committed in the system must have C_{latest} in its log. Since in `HandleCheckOldConfig`, we require $\mathit{GetMaxConfigIndex}(i) \leq \mathit{commitIndex}(i)$ to hold before the leader can append any newer config to its log, C_{new} can only be "one step" away from C_{latest}. □

Theorem 1. *There is at most one leader per term. This is the "Election Safety" property in Figure 3.2 and is proved for statically sized configurations in Lemma 2 of B.3 of [Ong14].*

$$\forall e, f \in \mathit{elections}$$
$$e.\mathit{eterm} = f.\mathit{eterm} \Rightarrow e.\mathit{eleader} = f.\mathit{eleader}$$

Proof. By Lemma 4 and Lemma 5, there can only be 3 possible configurations in the system at a time to form quorums:

$$C_{\text{latest}} = \{1, \ldots, n\}$$
$$C_{\text{new}+} = \{1, \ldots, n+1\}$$
$$C_{\text{new}-} = \{1, \ldots, n-1\}$$

For simplicity, assume the last server is the one that changes. Also note that if $n \geq 2$, all 3 are possible. If $n = 1$, only C_{latest} and $C_{\text{new}+}$ are possible.

- **Case 1.** $e.\mathit{evotes}, f.\mathit{evotes} \in \mathit{Quorum}(C_{\text{latest}})$.

  Because any two quorums of a config overlap, $e.\mathit{evotes} \cap f.\mathit{evotes} \neq \emptyset$. Suppose $s \in (e.\mathit{evotes} \cap f.\mathit{evotes})$. In `HandleRequestVoteRequest`,

  $$\begin{aligned} \mathit{grant} \triangleq\ & \land\ m.\mathit{mterm} = \mathit{currentTerm}[i]\ \textit{(1)} \\ & \land\ \mathit{logOk} \\ & \land\ \mathit{votedFor}[i] \in \{\mathit{Nil},\ j\}\ \textit{(3)} \end{aligned}$$

  Properties (1) and (3) guarantee that a server can only vote for at most one server per term.

  Since $s \in e.\mathit{evotes}$ and $s \in f.\mathit{evotes}$, $e.\mathit{eleader} = f.\mathit{eleader}$.

- **Case 2.** $e.evotes, f.evotes \in Quorum(C_{\text{new}+})$. Similar proof to **Case 1**.
- **Case 3.** $e.evotes, f.evotes \in Quorum(C_{\text{new}-})$. Similar proof to **Case 1**.
- **Case 4.** $e.evotes \in Quorum(C_{\text{latest}})$, $f.evotes \in Quorum(C_{\text{new}+})$.
  By Lemma 2, $e.evotes \cap f.evotes \neq \emptyset$. Afterwards, similar proof to **Case 1**.
- **Case 5.** $e.evotes \in Quorum(C_{\text{latest}})$, $f.evotes \in Quorum(C_{\text{new}-})$.
  By Lemma 1, $e.evotes \cap f.evotes \neq \emptyset$. Afterwards, similar proof to **Case 1**.
- **Case 6.** $e.evotes \in Quorum(C_{\text{new}+})$, $f.evotes \in Quorum(C_{\text{new}-})$.
  By Lemma 3, $e.evotes \cap f.evotes \neq \emptyset$. Afterwards, similar proof to **Case 1**.

Therefore, there is at most one leader per term. □

4.2 Proof Sketch for Availability: A leader can be elected in the future

One availability property of the system is that a leader is able to be elected in some future state from any state. Our proof sketch is to choose a server that has the most updated log. Then, this server can time out and cause a quorum of it's configuration to vote for it, which will always be able to happen because servers will vote if a candidate's log is up-to-date and the term is greater than theirs.

Other servers can also time out while this server times out. It is not harmful for another server to receive a majority of the votes and become leader, nor is a split vote harmful, since the randomized timeouts will not collide in future elections in practice.

5 Broken Raft?

5.1 Cluster membership changes

We present two possible edge cases during cluster membership changes that illustrate a possible area where Raft's description might be inconsistent. We could be misinterpreting the wording in [OO14, Ong14] and plan to send these cases to the author.

5.1.1 New servers need to vote for availability

Consider the following initial cluster, where s_1 is the leader, represented with the * and the log of each server is shown on the right. Note the log is a 3-tuple of the term it was appended, the type (configuration or value), and the contents.

s_1^*	(1, config, {1,2,3})
s_2	(1, config, {1,2,3})
s_3	(1, config, {1,2,3})
s_4	

s_1 gets a request to add s_4, so catches up s_4 with the config entry.

s_1^*	(1, config, {1,2,3})
s_2	(1, config, {1,2,3})
s_3	(1, config, {1,2,3})
s_4	(1, config, {1,2,3})

s_1 then appends a new config to its log to add s_4.

s_1^*	(1, config, {1,2,3}), (1, config, {1,2,3,4})
s_2	(1, config, {1,2,3})
s_3	(1, config, {1,2,3})
s_4	(1, config, {1,2,3})

s_3 dies and s_1 replicates the new config to s_2.

s_1^*	(1, config, {1,2,3}), (1, config, {1,2,3,4})
s_2	(1, config, {1,2,3}), (1, config, {1,2,3,4})
s_3	(1, config, {1,2,3})
s_4	(1, config, {1,2,3})

s_2 times out and starts an election and s_1 steps down. In this case, both s_1 and s_2 need s_4's vote to become the leader. Otherwise the system won't have a leader and is thus non-available.

5.1.2 New members voting causes inconsistencies

Consider the following situation with 4 initial servers and s_5 is added.

Use s^{*n} to denote a server being leader and s^{Tn} to denote a server timing out, both in term n.

s_1^{*1}	(1, config, {1,2,3,4})
s_2	(1, config, {1,2,3,4})
s_3	(1, config, {1,2,3,4})
s_4	(1, config, {1,2,3,4})
s_5	

s_1 catches up s_5.

s_1^{*1}	(1, config, {1,2,3,4})
s_2	(1, config, {1,2,3,4})
s_3	(1, config, {1,2,3,4})
s_4	(1, config, {1,2,3,4})
s_5	(1, config, {1,2,3,4})

s_1 appends new config.

s_1^{*1}	(1, config, {1,2,3,4}), (1,config,{1,2,3,4,5})
s_2	(1, config, {1,2,3,4})
s_3	(1, config, {1,2,3,4})
s_4	(1, config, {1,2,3,4})
s_5	(1, config, {1,2,3,4})

s_1 replicates new config to s_5.

s_1^{*1}	(1, config, {1,2,3,4}), (1,config,{1,2,3,4,5})
s_2	(1, config, {1,2,3,4})
s_3	(1, config, {1,2,3,4})
s_4	(1, config, {1,2,3,4})
s_5	(1, config, {1,2,3,4}), (1,config,{1,2,3,4,5})

s_1 dies temporarily.

s_1^{D*1}	(1, config, {1,2,3,4}), (1,config,{1,2,3,4,5})
s_2	(1, config, {1,2,3,4})
s_3	(1, config, {1,2,3,4})
s_4	(1, config, {1,2,3,4})
s_5	(1, config, {1,2,3,4}), (1,config,{1,2,3,4,5})

s_2 times out and starts an election.

s_1^{D*1}	(1, config, {1,2,3,4}), (1,config,{1,2,3,4,5})
s_2^{T2}	(1, config, {1,2,3,4})
s_3	(1, config, {1,2,3,4})
s_4	(1, config, {1,2,3,4})
s_5	(1, config, {1,2,3,4}), (1,config,{1,2,3,4,5})

s_2, s_3, s_4 vote for s_2. s_5 rejects. s_2 becomes leader.

s_1^{D*1}	(1, config, {1,2,3,4}), (1,config,{1,2,3,4,5})
s_2^{*2}	(1, config, {1,2,3,4})
s_3	(1, config, {1,2,3,4})
s_4	(1, config, {1,2,3,4})
s_5	(1, config, {1,2,3,4}), (1,config,{1,2,3,4,5})

s_2 appends a new config to its log.

s_1^{D*1}	(1, config, {1,2,3,4}), (1,config,{1,2,3,4,5})
s_2^{*2}	(1, config, {1,2,3,4}), (2, config, {2,3,4})
s_3	(1, config, {1,2,3,4})
s_4	(1, config, {1,2,3,4})
s_5	(1, config, {1,2,3,4}), (1,config,{1,2,3,4,5})

s_2 replicates new config to s_3 and **is committed**!

s_1^{D*1}	(1, config, {1,2,3,4}), (1,config,{1,2,3,4,5})
s_2^{*2}	(1, config, {1,2,3,4}), (2, config, {2,3,4})
s_3	(1, config, {1,2,3,4}), (2, config, {2,3,4})
s_4	(1, config, {1,2,3,4})
s_5	(1, config, {1,2,3,4}), (1,config,{1,2,3,4,5})

s_1 comes backs alive and times out and starts an election.

s_1^{T3}	(1, config, {1,2,3,4}), (1,config,{1,2,3,4,5})
s_2^{*2}	(1, config, {1,2,3,4}), (2, config, {2,3,4})
s_3	(1, config, {1,2,3,4}), (2, config, {2,3,4})
s_4	(1, config, {1,2,3,4})
s_5	(1, config, {1,2,3,4}), (1,config,{1,2,3,4,5})

If s_5 can vote, then s_1 can receive s_1, s_4, and s_5's votes and become the new leader. Then s_1 will try to replicate its log to everyone, including s_2 and s_3, which will conflict and overwrite the already committed entry (2, config, {2,3,4}) with an older uncommitted entry. This breaks the leader completeness property presented in Figure 3.2 of [Ong14]: "If a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms."

6 Conclusion and Future Work

We have presented a formal specification for Raft cluster membership changes and have proved that properties of the cluster are preserved during these changes. Future work involves further validating our modifications to the specification and modeling more invariants and properties of Raft. An interesting direction could be to study other formal verifications of Raft, such as Verdi's case study of Raft in PLDI 2015 [WWP+15].

References

[AS87] Bowen Alpern and Fred B Schneider. Recognizing safety and liveness. *Distributed computing*, 2(3):117–126, 1987.

[CDLM08] Kaustuv C Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. A tla+ proof system. *arXiv preprint arXiv:0811.1914*, 2008.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[Lam00] Leslie Lamport. A summary of TLA+. 2000.

[Lam02] Leslie Lamport. *Specifying systems: The TLA+ language and tools for hardware and software engineers.* Addison-Wesley Longman Publishing Co., Inc., 2002.

[Ong14] Diego Ongaro. *Consensus: Bridging theory and practice.* PhD thesis, Stanford University, 2014.

[OO14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Annual Technical Conference*, pages 305–320, 2014.

[raf] `https://raftconsensus.github.io/`.

[sec] `http://thesecretlivesofdata.com/raft/`.

[WWP+15] 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. *PLDI*, 2015.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.

A Original TLA+ Specification

Starts on next page.

MODULE *raft_orig*

This is the formal specification for the Raft consensus algorithm.

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *TLC*

The set of server *IDs*
CONSTANTS *Server*

The set of requests that can go into the *log*
CONSTANTS *Value*

Server states.
CONSTANTS *Follower*, *Candidate*, *Leader*

A reserved value.
CONSTANTS *Nil*

Message types:
CONSTANTS *RequestVoteRequest*, *RequestVoteResponse*,
AppendEntriesRequest, *AppendEntriesResponse*

Global variables

A bag of records representing requests and responses sent from one server to another. *TLAPS* doesn't support the Bags module, so this is a function mapping Message to *Nat*.
VARIABLE *messages*

A history variable used in the proof. This would not be present in an implementation.
Keeps track of successful elections, including the initial logs of the leader and voters' logs. Set of functions containing various things about successful elections (see *BecomeLeader*).
VARIABLE *elections*

A history variable used in the proof. This would not be present in an implementation.
Keeps track of every *log* ever in the system (set of logs).
VARIABLE *allLogs*

The following variables are all per server (functions with domain *Server*).

The server's term number.

VARIABLE *currentTerm*

The server's state (Follower, *Candidate*, or *Leader*).

VARIABLE *state*

The candidate the server voted for in its current term, or
Nil if it hasn't voted for any.

VARIABLE *votedFor*

$serverVars \triangleq \langle currentTerm, state, votedFor \rangle$

A Sequence of *log* entries. The index into this sequence is the index of the *log* entry. Unfortunately, the Sequence module defines *Head*(*s*) as the entry with index 1, so be careful not to use that!

VARIABLE *log*

The index of the latest entry in the *log* the state machine may apply.

VARIABLE *commitIndex*

$logVars \triangleq \langle log, commitIndex \rangle$

The following variables are used only on candidates:
The set of servers from which the candidate has received a *RequestVote* response in its *currentTerm*.

VARIABLE *votesResponded*

The set of servers from which the candidate has received a vote in its *currentTerm*.

VARIABLE *votesGranted*

A history variable used in the proof. This would not be present in an implementation.
Function from each server that voted for this candidate in its *currentTerm* to that voter's *log*.

VARIABLE *voterLog*

$candidateVars \triangleq \langle votesResponded, votesGranted, voterLog \rangle$

The following variables are used only on leaders:
The next entry to send to each follower.

VARIABLE *nextIndex*

The latest entry that each follower has acknowledged is the same as the leader's. This is used to calculate *commitIndex* on the leader.

VARIABLE *matchIndex*

$leaderVars \triangleq \langle nextIndex, matchIndex, elections \rangle$

End of per server variables.

All variables; used for stuttering (asserting state hasn't changed).

$vars \triangleq \langle messages, allLogs, serverVars, candidateVars, leaderVars, logVars \rangle$

Helpers

2

The set of all quorums. This just calculates simple majorities, but the only important property is that every quorum overlaps with every other.

$$Quorum \triangleq \{i \in \text{SUBSET } (Server) : Cardinality(i) * 2 > Cardinality(Server)\}$$

The term of the last entry in a *log*, or 0 if the *log* is empty.

$$LastTerm(xlog) \triangleq \text{IF } Len(xlog) = 0 \text{ THEN } 0 \text{ ELSE } xlog[Len(xlog)].term$$

Helper for *Send* and *Reply*. Given a message m and bag of messages, return a new bag of messages with one more m in it.

$$\begin{aligned}
&WithMessage(m, msgs) \triangleq \\
&\quad \text{IF } m \in \text{DOMAIN } msgs \text{ THEN} \\
&\qquad [msgs \text{ EXCEPT } ![m] = msgs[m] + 1] \\
&\quad \text{ELSE} \\
&\qquad msgs \mathbin{@@} (m :> 1)
\end{aligned}$$

Helper for *Discard* and *Reply*. Given a message m and bag of messages, return a new bag of messages with one less m in it.

$$\begin{aligned}
&WithoutMessage(m, msgs) \triangleq \\
&\quad \text{IF } m \in \text{DOMAIN } msgs \text{ THEN} \\
&\qquad [msgs \text{ EXCEPT } ![m] = msgs[m] - 1] \\
&\quad \text{ELSE} \\
&\qquad msgs
\end{aligned}$$

Add a message to the bag of messages.

$$Send(m) \triangleq messages' = WithMessage(m, messages)$$

Remove a message from the bag of messages. Used when a server is done processing a message.

$$Discard(m) \triangleq messages' = WithoutMessage(m, messages)$$

Combination of *Send* and *Discard*

$$\begin{aligned}
&Reply(response, request) \triangleq \\
&\quad messages' = WithoutMessage(request, WithMessage(response, messages))
\end{aligned}$$

Return the minimum value from a set, or undefined if the set is empty.

$$Min(s) \triangleq \text{CHOOSE } x \in s : \forall y \in s : x \leq y$$

Return the maximum value from a set, or undefined if the set is empty.

$$Max(s) \triangleq \text{CHOOSE } x \in s : \forall y \in s : x \geq y$$

Define initial values for all variables

$$\begin{aligned}
InitHistoryVars \triangleq\ & \land elections = \{\} \\
& \land allLogs = \{\} \\
& \land voterLog = [i \in Server \mapsto [j \in \{\} \mapsto \langle\rangle]] \\
InitServerVars \triangleq\ & \land currentTerm = [i \in Server \mapsto 1] \\
& \land state = [i \in Server \mapsto Follower] \\
& \land votedFor = [i \in Server \mapsto Nil]
\end{aligned}$$

$InitCandidateVars \triangleq \land votesResponded = [i \in Server \mapsto \{\}]$
$\qquad\qquad\qquad\qquad\quad \land votesGranted \quad = [i \in Server \mapsto \{\}]$

The values $nextIndex[i][i]$ and $matchIndex[i][i]$ are never read, since the leader does not send itself messages. It's still easier to include these in the functions.

$InitLeaderVars \triangleq \land nextIndex \quad = [i \in Server \mapsto [j \in Server \mapsto 1]]$
$\qquad\qquad\qquad\quad \land matchIndex = [i \in Server \mapsto [j \in Server \mapsto 0]]$

$InitLogVars \triangleq \land log \qquad\quad = [i \in Server \mapsto \langle\rangle]$
$\qquad\qquad\quad \land commitIndex = [i \in Server \mapsto 0]$

$Init \triangleq \land messages = [m \in \{\} \mapsto 0]$
$\qquad \land InitHistoryVars$
$\qquad \land InitServerVars$
$\qquad \land InitCandidateVars$
$\qquad \land InitLeaderVars$
$\qquad \land InitLogVars$

Define state transitions

Server i restarts from stable storage.
It loses everything but its $currentTerm$, $votedFor$, and log.

$Restart(i) \triangleq$
$\quad \land \; state' \qquad\qquad = [state \text{ EXCEPT } ![i] = Follower]$
$\quad \land \; votesResponded' = [votesResponded \text{ EXCEPT } ![i] = \{\}]$
$\quad \land \; votesGranted' \quad = [votesGranted \text{ EXCEPT } ![i] = \{\}]$
$\quad \land \; voterLog' \qquad = [voterLog \text{ EXCEPT } ![i] = [j \in \{\} \mapsto \langle\rangle]]$
$\quad \land \; nextIndex' \qquad = [nextIndex \text{ EXCEPT } ![i] = [j \in Server \mapsto 1]]$
$\quad \land \; matchIndex' \quad = [matchIndex \text{ EXCEPT } ![i] = [j \in Server \mapsto 0]]$
$\quad \land \; commitIndex' \quad = [commitIndex \text{ EXCEPT } ![i] = 0]$
$\quad \land \; \text{UNCHANGED } \langle messages, currentTerm, votedFor, log, elections\rangle$

Server i times out and starts a new election.

$Timeout(i) \triangleq \land state[i] \in \{Follower, Candidate\}$
$\qquad\qquad\quad \land state' = [state \text{ EXCEPT } ![i] = Candidate]$
$\qquad\qquad\quad \land currentTerm' = [currentTerm \text{ EXCEPT } ![i] = currentTerm[i] + 1]$

Most implementations would probably just set the local vote atomically, but messaging localhost for it is weaker.

$\qquad\qquad\quad \land votedFor' = [votedFor \text{ EXCEPT } ![i] = Nil]$
$\qquad\qquad\quad \land votesResponded' = [votesResponded \text{ EXCEPT } ![i] = \{\}]$
$\qquad\qquad\quad \land votesGranted' \quad = [votesGranted \text{ EXCEPT } ![i] = \{\}]$
$\qquad\qquad\quad \land voterLog' \qquad = [voterLog \text{ EXCEPT } ![i] = [j \in \{\} \mapsto \langle\rangle]]$
$\qquad\qquad\quad \land \text{UNCHANGED } \langle messages, leaderVars, logVars\rangle$

Candidate i sends j a $RequestVote$ request.

$RequestVote(i, j) \triangleq$
$\quad \land state[i] = Candidate$

$\land\ j \notin votesResponded[i]$
$\land\ Send([mtype \mapsto RequestVoteRequest,$
$\quad mterm \mapsto currentTerm[i],$
$\quad mlastLogTerm \mapsto LastTerm(log[i]),$
$\quad mlastLogIndex \mapsto Len(log[i]),$
$\quad msource \mapsto i,$
$\quad mdest \mapsto j])$
$\land\ \text{UNCHANGED}\ \langle serverVars, candidateVars, leaderVars, logVars\rangle$

Leader i sends j an *AppendEntries* request containing up to 1 entry.
While implementations may want to send more than 1 at a time, this spec uses just 1 because it minimizes atomic regions without loss of generality.

$AppendEntries(i, j) \triangleq$
$\land\ i \neq j$
$\land\ state[i] = Leader$
$\land\ \text{LET}\ prevLogIndex \triangleq nextIndex[i][j] - 1$
$\quad prevLogTerm \triangleq \text{IF}\ prevLogIndex > 0\ \text{THEN}$
$\qquad log[i][prevLogIndex].term$
$\quad \text{ELSE}$
$\qquad 0$

Send up to 1 entry, constrained by the end of the *log*.

$\quad lastEntry \triangleq Min(\{Len(log[i]), nextIndex[i][j]\})$
$\quad entries \triangleq SubSeq(log[i], nextIndex[i][j], lastEntry)$
$\text{IN}\quad Send([mtype \mapsto AppendEntriesRequest,$
$\quad mterm \mapsto currentTerm[i],$
$\quad mprevLogIndex \mapsto prevLogIndex,$
$\quad mprevLogTerm \mapsto prevLogTerm,$
$\quad mentries \mapsto entries,$

mlog is used as a history variable for the proof.
It would not exist in a real implementation.

$\quad mlog \mapsto log[i],$
$\quad mcommitIndex \mapsto Min(\{commitIndex[i], lastEntry\}),$
$\quad msource \mapsto i,$
$\quad mdest \mapsto j])$
$\land\ \text{UNCHANGED}\ \langle serverVars, candidateVars, leaderVars, logVars\rangle$

Candidate i transitions to leader.

$BecomeLeader(i) \triangleq$
$\land\ state[i] = Candidate$
$\land\ votesGranted[i] \in Quorum$
$\land\ state' = [state\ \text{EXCEPT}\ ![i] = Leader]$
$\land\ nextIndex' = [nextIndex\ \text{EXCEPT}\ ![i] =$
$\qquad [j \in Server \mapsto Len(log[i]) + 1]]$
$\land\ matchIndex' = [matchIndex\ \text{EXCEPT}\ ![i] =$
$\qquad [j \in Server \mapsto 0]]$

```
    ∧ elections′    = elections ∪
                         {[eterm      ↦ currentTerm[i],
                           eleader    ↦ i,
                           elog       ↦ log[i],
                           evotes     ↦ votesGranted[i],
                           evoterLog  ↦ voterLog[i]]}
    ∧ UNCHANGED ⟨messages, currentTerm, votedFor, candidateVars, logVars⟩
```

Leader i receives a client request to add v to the *log*.

```
ClientRequest(i, v) ≜
    ∧ state[i] = Leader
    ∧ LET entry ≜ [term  ↦ currentTerm[i],
                   value ↦ v]
          newLog ≜ Append(log[i], entry)
      IN  log′ = [log EXCEPT ![i] = newLog]
    ∧ UNCHANGED ⟨messages, serverVars, candidateVars,
                   leaderVars, commitIndex⟩
```

Leader i advances its *commitIndex*.
This is done as a separate step from handling *AppendEntries* responses,
in part to minimize atomic regions, and in part so that leaders of
single-server clusters are able to mark entries committed.

```
AdvanceCommitIndex(i) ≜
    ∧ state[i] = Leader
    ∧ LET  The set of servers that agree up through index.
          Agree(index) ≜ {i} ∪ {k ∈ Server :
                                    matchIndex[i][k] ≥ index}
           The maximum indexes for which a quorum agrees
          agreeIndexes ≜ {index ∈ 1 . . Len(log[i]) :
                                Agree(index) ∈ Quorum}
           New value for commitIndex′[i]
          newCommitIndex ≜
             IF  ∧ agreeIndexes ≠ {}
                 ∧ log[i][Max(agreeIndexes)].term = currentTerm[i]
              THEN
                 Max(agreeIndexes)
              ELSE
                 commitIndex[i]
      IN  commitIndex′ = [commitIndex EXCEPT ![i] = newCommitIndex]
    ∧ UNCHANGED ⟨messages, serverVars, candidateVars, leaderVars, log⟩
```

Message handlers
$i =$ recipient, $j =$ sender, $m =$ message

Server i receives a *RequestVote* request from server j with

$m.mterm \leq currentTerm[i]$.

```
HandleRequestVoteRequest(i, j, m) ≜
  LET logOk ≜ ∨ m.mlastLogTerm > LastTerm(log[i])
              ∨ ∧ m.mlastLogTerm = LastTerm(log[i])
                ∧ m.mlastLogIndex ≥ Len(log[i])
      grant ≜ ∧ m.mterm = currentTerm[i]
              ∧ logOk
              ∧ votedFor[i] ∈ {Nil, j}
  IN  ∧ m.mterm ≤ currentTerm[i]
      ∧ ∨ grant  ∧ votedFor′ = [votedFor EXCEPT ![i] = j]
        ∨ ¬grant ∧ UNCHANGED votedFor
      ∧ Reply([mtype        ↦ RequestVoteResponse,
               mterm        ↦ currentTerm[i],
               mvoteGranted ↦ grant,
                 mlog is used just for the elections history variable for
                 the proof. It would not exist in a real implementation.
               mlog         ↦ log[i],
               msource      ↦ i,
               mdest        ↦ j],
               m)
      ∧ UNCHANGED ⟨state, currentTerm, candidateVars, leaderVars, logVars⟩
```

Server i receives a *RequestVote* response from server j with $m.mterm = currentTerm[i]$.

```
HandleRequestVoteResponse(i, j, m) ≜
    This tallies votes even when the current state is not Candidate, but
    they won't be looked at, so it doesn't matter.
  ∧ m.mterm = currentTerm[i]
  ∧ votesResponded′ = [votesResponded EXCEPT ![i] =
                            votesResponded[i] ∪ {j}]
  ∧ ∨ ∧ m.mvoteGranted
      ∧ votesGranted′ = [votesGranted EXCEPT ![i] =
                              votesGranted[i] ∪ {j}]
      ∧ voterLog′ = [voterLog EXCEPT ![i] =
                          voterLog[i] @@ (j :> m.mlog)]
    ∨ ∧ ¬m.mvoteGranted
      ∧ UNCHANGED ⟨votesGranted, voterLog⟩
  ∧ Discard(m)
  ∧ UNCHANGED ⟨serverVars, votedFor, leaderVars, logVars⟩
```

Server i receives an *AppendEntries* request from server j with $m.mterm \leq currentTerm[i]$. This just handles $m.entries$ of length 0 or 1, but implementations could safely accept more by treating them the same as multiple independent requests of 1 entry.

```
HandleAppendEntriesRequest(i, j, m) ≜
```

```
LET logOk ≜ ∨ m.mprevLogIndex = 0
            ∨ ∧ m.mprevLogIndex > 0
              ∧ m.mprevLogIndex ≤ Len(log[i])
              ∧ m.mprevLogTerm = log[i][m.mprevLogIndex].term
IN  ∧ m.mterm ≤ currentTerm[i]
    ∧ ∨ ∧  reject request
          ∨ m.mterm < currentTerm[i]
          ∨ ∧ m.mterm = currentTerm[i]
            ∧ state[i] = Follower
            ∧ ¬logOk
        ∧ Reply([mtype          ↦ AppendEntriesResponse,
                 mterm          ↦ currentTerm[i],
                 msuccess       ↦ FALSE,
                 mmatchIndex    ↦ 0,
                 msource        ↦ i,
                 mdest          ↦ j],
                 m)
        ∧ UNCHANGED ⟨serverVars, logVars⟩
      ∨  return to follower state
        ∧ m.mterm = currentTerm[i]
        ∧ state[i] = Candidate
        ∧ state′ = [state EXCEPT ![i] = Follower]
        ∧ UNCHANGED ⟨currentTerm, votedFor, logVars, messages⟩
      ∨  accept request
        ∧ m.mterm = currentTerm[i]
        ∧ state[i] = Follower
        ∧ logOk
        ∧ LET index ≜ m.mprevLogIndex + 1
          IN  ∨  already done with request
                ∧ ∨ m.mentries = ⟨⟩
                  ∨ ∧ Len(log[i]) ≥ index
                    ∧ log[i][index].term = m.mentries[1].term
                  This could make our commitIndex decrease (for
                  example if we process an old, duplicated request),
                  but that doesn't really affect anything.
                ∧ commitIndex′ = [commitIndex EXCEPT ![i] =
                                        m.mcommitIndex]
                ∧ Reply([mtype          ↦ AppendEntriesResponse,
                         mterm          ↦ currentTerm[i],
                         msuccess       ↦ TRUE,
                         mmatchIndex    ↦ m.mprevLogIndex +
                                            Len(m.mentries),
                         msource        ↦ i,
                         mdest          ↦ j],
                         m)
```

$\land$ UNCHANGED $\langle serverVars, logVars \rangle$

$\lor$ conflict: remove 1 entry

$\land\ m.mentries \neq \langle\rangle$

$\land\ Len(log[i]) \geq index$

$\land\ log[i][index].term \neq m.mentries[1].term$

$\land$ LET $new \triangleq [index2 \in 1 \, .. \, (Len(log[i]) - 1) \mapsto log[i][index2]]$

IN $log' = [log$ EXCEPT $![i] = new]$

$\land$ UNCHANGED $\langle serverVars, commitIndex, messages \rangle$

$\lor$ no conflict: append entry

$\land\ m.mentries \neq \langle\rangle$

$\land\ Len(log[i]) = m.mprevLogIndex$

$\land\ log' = [log$ EXCEPT $![i] = Append(log[i], m.mentries[1])]$

$\land$ UNCHANGED $\langle serverVars, commitIndex, messages \rangle$

$\land$ UNCHANGED $\langle candidateVars, leaderVars \rangle$

Server i receives an *AppendEntries* response from server j with $m.mterm = currentTerm[i]$.

$HandleAppendEntriesResponse(i, j, m) \triangleq$

$\land\ m.mterm = currentTerm[i]$

$\land \lor \land\ m.msuccess$ successful

$\land\ nextIndex' = [nextIndex$ EXCEPT $![i][j] = m.mmatchIndex + 1]$

$\land\ matchIndex' = [matchIndex$ EXCEPT $![i][j] = m.mmatchIndex]$

$\lor \land \neg m.msuccess$ not successful

$\land\ nextIndex' = [nextIndex$ EXCEPT $![i][j] = Max(\{nextIndex[i][j] - 1, 1\})]$

$\land$ UNCHANGED $\langle matchIndex \rangle$

$\land\ Discard(m)$

$\land$ UNCHANGED $\langle serverVars, candidateVars, logVars, elections \rangle$

Any *RPC* with a newer term causes the recipient to advance its term first.

$UpdateTerm(i, j, m) \triangleq$

$\land\ m.mterm > currentTerm[i]$

$\land\ currentTerm' = [currentTerm$ EXCEPT $![i] = m.mterm]$

$\land\ state' = [state$ EXCEPT $![i] = Follower]$

$\land\ votedFor' = [votedFor$ EXCEPT $![i] = Nil]$

messages is unchanged so m can be processed further.

$\land$ UNCHANGED $\langle messages, candidateVars, leaderVars, logVars \rangle$

Responses with stale terms are ignored.

$DropStaleResponse(i, j, m) \triangleq$

$\land\ m.mterm < currentTerm[i]$

$\land\ Discard(m)$

$\land$ UNCHANGED $\langle serverVars, candidateVars, leaderVars, logVars \rangle$

Receive a message.

$Receive(m) \triangleq$

$\quad$ LET $i \triangleq m.mdest$

$\qquad\quad j \triangleq m.msource$

$\quad$ IN $\quad$ Any RPC with a newer term causes the recipient to advance its term first. Responses with stale terms are ignored.

$$
\begin{array}{l}
\lor\ UpdateTerm(i, j, m) \\
\lor\ \land\ m.mtype = RequestVoteRequest \\
\quad\ \land\ HandleRequestVoteRequest(i, j, m) \\
\lor\ \land\ m.mtype = RequestVoteResponse \\
\quad\ \land\ \lor\ DropStaleResponse(i, j, m) \\
\qquad\ \ \lor\ HandleRequestVoteResponse(i, j, m) \\
\lor\ \land\ m.mtype = AppendEntriesRequest \\
\quad\ \land\ HandleAppendEntriesRequest(i, j, m) \\
\lor\ \land\ m.mtype = AppendEntriesResponse \\
\quad\ \land\ \lor\ DropStaleResponse(i, j, m) \\
\qquad\ \ \lor\ HandleAppendEntriesResponse(i, j, m)
\end{array}
$$

End of message handlers.

Network state transitions

The network duplicates a message

$$
\begin{array}{l}
DuplicateMessage(m) \triangleq \\
\quad \land\ Send(m) \\
\quad \land\ \text{UNCHANGED}\ \langle serverVars, candidateVars, leaderVars, logVars \rangle
\end{array}
$$

The network drops a message

$$
\begin{array}{l}
DropMessage(m) \triangleq \\
\quad \land\ Discard(m) \\
\quad \land\ \text{UNCHANGED}\ \langle serverVars, candidateVars, leaderVars, logVars \rangle
\end{array}
$$

Defines how the variables may transition.

$$
\begin{array}{l}
Next \triangleq\ \land\ \lor\ \exists\, i \in Server : Restart(i) \\
\qquad\qquad\ \lor\ \exists\, i \in Server : Timeout(i) \\
\qquad\qquad\ \lor\ \exists\, i, j \in Server : RequestVote(i, j) \\
\qquad\qquad\ \lor\ \exists\, i \in Server : BecomeLeader(i) \\
\qquad\qquad\ \lor\ \exists\, i \in Server, v \in Value : ClientRequest(i, v) \\
\qquad\qquad\ \lor\ \exists\, i \in Server : AdvanceCommitIndex(i) \\
\qquad\qquad\ \lor\ \exists\, i, j \in Server : AppendEntries(i, j) \\
\qquad\qquad\ \lor\ \exists\, m \in \text{DOMAIN}\ messages : Receive(m) \\
\qquad\qquad\ \lor\ \exists\, m \in \text{DOMAIN}\ messages : DuplicateMessage(m) \\
\qquad\qquad\ \lor\ \exists\, m \in \text{DOMAIN}\ messages : DropMessage(m)
\end{array}
$$

History variable that tracks every log ever:

$$\land\ allLogs' = allLogs \cup \{log[i] : i \in Server\}$$

The specification must start with the initial state and transition according to *Next*.

$Spec \triangleq Init \land \Box[Next]_{vars}$

\ ∗ *Changelog*:
\ ∗
\ ∗ 2014 − 12 − 02:
\ ∗ − Fix *AppendEntries* to only send one entry at a time, as originally
\ ∗ intended. Since *SubSeq* is inclusive, the upper bound of the range should
\ ∗ have been *nextIndex*, not $nextIndex + 1$. Thanks to *Igor Kovalenko* for
\ ∗ reporting the issue.
\ ∗ − Change $matchIndex'$ to *matchIndex* (without the apostrophe) in
\ ∗ *AdvanceCommitIndex*. This apostrophe was not intentional and perhaps
\ ∗ confusing, though it makes no practical difference ($matchIndex'$ equals
\ ∗ *matchIndex*). Thanks to *Hugues Evrard* for reporting the issue.
\ ∗
\ ∗ 2014 − 07 − 06:
\ ∗ − Version from *PhD* dissertation

B Our Modified TLA+ Specification

Starts on next page.

MODULE *raft*

This is the formal specification for the Raft consensus algorithm.

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *TLC*

The number of rounds to catch new servers up for.
Must be ≥ 1.

CONSTANTS *NumRounds*

The initial and global set of servers.

CONSTANTS *InitServer*, *Server*

Log metadata to distinguish values from configuration changes.

CONSTANT *ValueEntry*, *ConfigEntry*

The set of values that can go into the *log*.

CONSTANTS *Value*

Server states.

CONSTANTS *Follower*, *Candidate*, *Leader*

A reserved value.

CONSTANTS *Nil*

Message types:

CONSTANTS *RequestVoteRequest*, *RequestVoteResponse*,
AppendEntriesRequest, *AppendEntriesResponse*,
CatchupRequest, *CatchupResponse*,
CheckOldConfig

Global variables

A bag of records representing requests and responses sent from one server to another. *TLAPS* doesn't support the Bags module, so this is a function mapping Message to *Nat*.

VARIABLE *messages*

A history variable used in the proof. This would not be present in an implementation.
Keeps track of successful elections, including the initial logs of the

leader and voters' logs. Set of functions containing various things about successful elections (see *BecomeLeader*).

VARIABLE *elections*

A history variable used in the proof. This would not be present in an implementation.

Keeps track of every *log* ever in the system (set of logs).

VARIABLE *allLogs*

The following variables are all per server (functions with domain *Server*).

The server's term number.

VARIABLE *currentTerm*

The server's state (Follower, *Candidate*, or *Leader*).

VARIABLE *state*

The candidate the server voted for in its current term, or Nil if it hasn't voted for any.

VARIABLE *votedFor*

$serverVars \triangleq \langle currentTerm, state, votedFor \rangle$

A Sequence of *log* entries. The index into this sequence is the index of the *log* entry. Unfortunately, the Sequence module defines $Head(s)$ as the entry with index 1, so be careful not to use that!

VARIABLE *log*

The index of the latest entry in the *log* the state machine may apply.

VARIABLE *commitIndex*

$logVars \triangleq \langle log, commitIndex \rangle$

The following variables are used only on candidates:

The set of servers from which the candidate has received a *RequestVote* response in its *currentTerm*.

VARIABLE *votesResponded*

The set of *Server* from which the candidate has received a vote in its *currentTerm*.

VARIABLE *votesGranted*

A history variable used in the proof. This would not be present in an implementation.

Function from each server that voted for this candidate in its *currentTerm* to that voter's *log*.

VARIABLE *voterLog*

$candidateVars \triangleq \langle votesResponded, votesGranted, voterLog \rangle$

The following variables are used only on leaders:

The next entry to send to each follower.

VARIABLE *nextIndex*

The latest entry that each follower has acknowledged is the same as the leader's. This is used to calculate *commitIndex* on the leader.

VARIABLE $matchIndex$

$leaderVars \triangleq \langle nextIndex, matchIndex, elections \rangle$

End of per server variables.

All variables; used for stuttering (asserting state hasn't changed).

$vars \triangleq \langle messages, allLogs, serverVars, candidateVars,$
$\quad leaderVars, logVars \rangle$

Helpers

The set of all quorums for a server configuration.
This just calculates simple majorities, but the only
important property is that every quorum overlaps with every other.

$Quorum(config) \triangleq \{i \in \text{SUBSET } (config) : Cardinality(i) * 2 > Cardinality(config)\}$

The term of the last entry in a *log*, or 0 if the *log* is empty.

$LastTerm(xlog) \triangleq \text{IF } Len(xlog) = 0 \text{ THEN } 0 \text{ ELSE } xlog[Len(xlog)].term$

Helper for *Send* and *Reply*. Given a message *m* and bag of messages, return a new bag of messages with one more *m* in it.

$WithMessage(m, msgs) \triangleq$
$\quad \text{IF } m \in \text{DOMAIN } msgs \text{ THEN}$
$\quad\quad [msgs \text{ EXCEPT } ![m] = msgs[m] + 1]$
$\quad \text{ELSE}$
$\quad\quad msgs \,@@\, (m :> 1)$

Helper for *Discard* and *Reply*. Given a message *m* and bag of messages, return a new bag of messages with one less *m* in it.

$WithoutMessage(m, msgs) \triangleq$
$\quad \text{IF } m \in \text{DOMAIN } msgs \text{ THEN}$
$\quad\quad [msgs \text{ EXCEPT } ![m] = msgs[m] - 1]$
$\quad \text{ELSE}$
$\quad\quad msgs$

Add a message to the bag of messages.

$Send(m) \triangleq messages' = WithMessage(m, messages)$

Remove a message from the bag of messages. Used when a server is done processing a message.

$Discard(m) \triangleq messages' = WithoutMessage(m, messages)$

Combination of *Send* and *Discard*

$Reply(response, request) \triangleq$

$messages' = WithoutMessage(request, WithMessage(response, messages))$

Return the minimum value from a set, or undefined if the set is empty.

$Min(s) \triangleq \text{CHOOSE } x \in s : \forall y \in s : x \leq y$

Return the maximum value from a set, or undefined if the set is empty.

$Max(s) \triangleq \text{CHOOSE } x \in s : \forall y \in s : x \geq y$

Return the index of the latest configuration in server i's log.

$$
\begin{aligned}
& GetMaxConfigIndex(i) \triangleq \\
& \quad \text{LET } configIndexes \triangleq \{index \in 1 \,..\, Len(log[i]) : log[i][index].type = ConfigEntry\} \\
& \quad \text{IN } \quad \text{IF } configIndexes = \{\} \text{ THEN } 0 \\
& \qquad\qquad \text{ELSE } Max(configIndexes)
\end{aligned}
$$

Return the configuration of teh latest configuration in server i's log.

$$
\begin{aligned}
& GetConfig(i) \triangleq \\
& \quad \text{IF } GetMaxConfigIndex(i) = 0 \text{ THEN } InitServer \\
& \quad \text{ELSE } log[i][GetMaxConfigIndex(i)].value
\end{aligned}
$$

Define initial values for all variables

$$
\begin{aligned}
InitHistoryVars \triangleq\ & \land elections = \{\} \\
& \land allLogs = \{\} \\
& \land voterLog = [i \in Server \mapsto [j \in \{\} \mapsto \langle\rangle]] \\
InitServerVars \triangleq\ & \land currentTerm = [i \in Server \mapsto 1] \\
& \land state = [i \in Server \mapsto Follower] \\
& \land votedFor = [i \in Server \mapsto Nil] \\
InitCandidateVars \triangleq\ & \land votesResponded = [i \in Server \mapsto \{\}] \\
& \land votesGranted = [i \in Server \mapsto \{\}]
\end{aligned}
$$

The values $nextIndex[i][i]$ and $matchIndex[i][i]$ are never read, since the leader does not send itself messages. It's still easier to include these in the functions.

$$
\begin{aligned}
InitLeaderVars \triangleq\ & \land nextIndex = [i \in Server \mapsto [j \in Server \mapsto 1]] \\
& \land matchIndex = [i \in Server \mapsto [j \in Server \mapsto 0]] \\
InitLogVars \triangleq\ & \land log = [i \in Server \mapsto \langle\rangle] \\
& \land commitIndex = [i \in Server \mapsto 0] \\
Init \triangleq\ & \land messages = [m \in \{\} \mapsto 0] \\
& \land InitHistoryVars \\
& \land InitServerVars \\
& \land InitCandidateVars \\
& \land InitLeaderVars \\
& \land InitLogVars
\end{aligned}
$$

Define state transitions

Server i restarts from stable storage.
It loses everything but its *currentTerm*, *votedFor*, and *log*.

$$
\begin{array}{l}
Restart(i) \triangleq \\
\quad \land\ i \in GetConfig(i) \\
\quad \land\ state' = [state \text{ EXCEPT } ![i] = Follower] \\
\quad \land\ votesResponded' = [votesResponded \text{ EXCEPT } ![i] = \{\}] \\
\quad \land\ votesGranted' \quad = [votesGranted \text{ EXCEPT } ![i] = \{\}] \\
\quad \land\ voterLog' \qquad = [voterLog \text{ EXCEPT } ![i] = [j \in \{\} \mapsto \langle\rangle]] \\
\quad \land\ nextIndex' \qquad = [nextIndex \text{ EXCEPT } ![i] = [j \in Server \mapsto 1]] \\
\quad \land\ matchIndex' \quad = [matchIndex \text{ EXCEPT } ![i] = [j \in Server \mapsto 0]] \\
\quad \land\ commitIndex' \quad = [commitIndex \text{ EXCEPT } ![i] = 0] \\
\quad \land\ \text{UNCHANGED } \langle messages, currentTerm, votedFor, log, elections\rangle
\end{array}
$$

Server i times out and starts a new election.

$$
\begin{array}{ll}
Timeout(i) \triangleq & \land\ state[i] \in \{Follower, Candidate\} \\
& \land\ i \in GetConfig(i) \\
& \land\ state' = [state \text{ EXCEPT } ![i] = Candidate] \\
& \land\ currentTerm' = [currentTerm \text{ EXCEPT } ![i] = currentTerm[i] + 1]
\end{array}
$$

Most implementations would probably just set the local vote
atomically, but messaging localhost for it is weaker.

$$
\begin{array}{ll}
\qquad\qquad\qquad & \land\ votedFor' = [votedFor \text{ EXCEPT } ![i] = Nil] \\
& \land\ votesResponded' = [votesResponded \text{ EXCEPT } ![i] = \{\}] \\
& \land\ votesGranted' \quad = [votesGranted \text{ EXCEPT } ![i] = \{\}] \\
& \land\ voterLog' \qquad = [voterLog \text{ EXCEPT } ![i] = [j \in \{\} \mapsto \langle\rangle]] \\
& \land\ \text{UNCHANGED } \langle messages, leaderVars, logVars\rangle
\end{array}
$$

Candidate i sends j a *RequestVote* request.

$$
\begin{array}{l}
RequestVote(i, j) \triangleq \\
\quad \land\ state[i] = Candidate \\
\quad \land\ j \in (GetConfig(i) \setminus votesResponded[i]) \\
\quad \land\ Send([\begin{array}[t]{ll} mtype & \mapsto RequestVoteRequest, \\ mterm & \mapsto currentTerm[i], \\ mlastLogTerm & \mapsto LastTerm(log[i]), \\ mlastLogIndex & \mapsto Len(log[i]), \\ msource & \mapsto i, \\ mdest & \mapsto j]) \end{array} \\
\quad \land\ \text{UNCHANGED } \langle serverVars, candidateVars, leaderVars, logVars\rangle
\end{array}
$$

Leader i sends j an *AppendEntries* request containing up to 1 entry.
While implementations may want to send more than 1 at a time, this spec uses
just 1 because it minimizes atomic regions without loss of generality.

$$
\begin{array}{l}
AppendEntries(i, j) \triangleq \\
\quad \land\ i \neq j \\
\quad \land\ state[i] = Leader \\
\quad \land\ j \in GetConfig(i) \\
\quad \land\ \text{LET } prevLogIndex \triangleq nextIndex[i][j] - 1
\end{array}
$$

```
      prevLogTerm ≜ IF prevLogIndex > 0 THEN
                        log[i][prevLogIndex].term
                     ELSE
                        0
       Send up to 1 entry, constrained by the end of the log.
      lastEntry ≜ Min({Len(log[i]), nextIndex[i][j]})
      entries ≜ SubSeq(log[i], nextIndex[i][j], lastEntry)
   IN Send([mtype          ↦ AppendEntriesRequest,
            mterm          ↦ currentTerm[i],
            mprevLogIndex  ↦ prevLogIndex,
            mprevLogTerm   ↦ prevLogTerm,
            mentries       ↦ entries,
             mlog is used as a history variable for the proof.
             It would not exist in a real implementation.
            mlog           ↦ log[i],
            mcommitIndex   ↦ Min({commitIndex[i], lastEntry}),
            msource        ↦ i,
            mdest          ↦ j])
  ∧ UNCHANGED ⟨serverVars, candidateVars, leaderVars, logVars⟩

 Candidate i transitions to leader.
BecomeLeader(i) ≜
  ∧ state[i] = Candidate
  ∧ votesGranted[i] ∈ Quorum(GetConfig(i))
  ∧ state'      = [state EXCEPT ![i] = Leader]
  ∧ nextIndex'  = [nextIndex EXCEPT ![i] =
                      [j ∈ Server ↦ Len(log[i]) + 1]]
  ∧ matchIndex' = [matchIndex EXCEPT ![i] =
                      [j ∈ Server ↦ 0]]
  ∧ elections'  = elections ∪
                      {[eterm      ↦ currentTerm[i],
                        eleader    ↦ i,
                        elog       ↦ log[i],
                        evotes     ↦ votesGranted[i],
                        evoterLog  ↦ voterLog[i]]}
  ∧ UNCHANGED ⟨messages, currentTerm, votedFor, candidateVars, logVars⟩

 Leader i receives a client request to add v to the log.
ClientRequest(i, v) ≜
  ∧ state[i] = Leader
  ∧ LET entry ≜ [term  ↦ currentTerm[i],
                 type  ↦ ValueEntry,
                 value ↦ v]
        newLog ≜ Append(log[i], entry)
    IN  log' = [log EXCEPT ![i] = newLog]
```

$\land$ UNCHANGED $\langle messages, serverVars, candidateVars, leaderVars, commitIndex\rangle$

Leader i advances its *commitIndex*.
This is done as a separate step from handling *AppendEntries* responses, in part to minimize atomic regions, and in part so that leaders of single-server clusters are able to mark entries committed.

$AdvanceCommitIndex(i) \triangleq$

$\quad \land state[i] = Leader$

$\quad \land$ LET The set of servers that agree up through index.

$\qquad\quad Agree(index) \triangleq \{i\} \cup \{k \in GetConfig(i) : matchIndex[i][k] \geq index\}$

$\qquad\quad$ The maximum indexes for which a quorum agrees

$\qquad\quad agreeIndexes \triangleq \{index \in 1 \,..\, Len(log[i]) : Agree(index) \in Quorum(GetConfig(i))\}$

$\qquad\quad$ New value for $commitIndex'[i]$

$\qquad\quad newCommitIndex \triangleq$

$\qquad\qquad$ IF $\land agreeIndexes \neq \{\}$

$\qquad\qquad\quad \land log[i][Max(agreeIndexes)].term = currentTerm[i]$

$\qquad\qquad$ THEN

$\qquad\qquad\quad Max(agreeIndexes)$

$\qquad\qquad$ ELSE

$\qquad\qquad\quad commitIndex[i]$

$\qquad$ IN $\quad commitIndex' = [commitIndex$ EXCEPT $![i] = newCommitIndex]$

$\quad \land$ UNCHANGED $\langle messages, serverVars, candidateVars, leaderVars, log\rangle$

Leader i adds a new server j to the cluster.

$AddNewServer(i, j) \triangleq$

$\quad \land state[i] = Leader$

$\quad \land j \notin GetConfig(i)$

$\quad \land currentTerm' = [currentTerm$ EXCEPT $![j] = 1]$

$\quad \land votedFor' = [votedFor$ EXCEPT $![j] = Nil]$

$\quad \land Send([mtype \mapsto CatchupRequest,$

$\qquad\qquad mterm \mapsto currentTerm[i],$

$\qquad\qquad mlogLen \mapsto matchIndex[i][j],$

$\qquad\qquad mentries \mapsto SubSeq(log[i], nextIndex[i][j], commitIndex[i]),$

$\qquad\qquad mcommitIndex \mapsto commitIndex[i],$

$\qquad\qquad msource \mapsto i,$

$\qquad\qquad mdest \mapsto j,$

$\qquad\qquad mrounds \mapsto NumRounds])$

$\quad \land$ UNCHANGED $\langle state, leaderVars, logVars, candidateVars\rangle$

Leader i removes a server j (possibly itself) from the cluster.

$DeleteServer(i, j) \triangleq$

$\quad \land state[i] = Leader$

$\land\ state[j] \in \{Follower,\ Candidate\}$
$\land\ j \in GetConfig(i)$
$\land\ j \neq i$ *TODO*: A leader cannot remove itself.
$\land\ Send([mtype \mapsto CheckOldConfig,$
$\quad mterm \mapsto currentTerm[i],$
$\quad madd \mapsto \text{FALSE},$
$\quad mserver \mapsto j,$
$\quad msource \mapsto i,$
$\quad mdest \mapsto i])$
$\land\ \text{UNCHANGED}\ \langle serverVars,\ candidateVars,\ leaderVars,\ logVars\rangle$

Message handlers
$i =$ recipient, $j =$ sender, $m =$ message

Server i receives a *RequestVote* request from server j with $m.mterm \leq currentTerm[i]$.

$HandleRequestVoteRequest(i,\ j,\ m) \triangleq$
$\quad \text{LET}\ logOk \triangleq \lor\ m.mlastLogTerm > LastTerm(log[i])$
$\qquad\qquad \lor\ \land\ m.mlastLogTerm = LastTerm(log[i])$
$\qquad\qquad\quad \land\ m.mlastLogIndex \geq Len(log[i])$
$\qquad grant \triangleq \land\ m.mterm = currentTerm[i]$
$\qquad\qquad \land\ logOk$
$\qquad\qquad \land\ votedFor[i] \in \{Nil,\ j\}$
$\quad \text{IN}\ \land\ m.mterm \leq currentTerm[i]$
$\qquad \land\ \lor\ grant\ \land\ votedFor' = [votedFor\ \text{EXCEPT}\ ![i] = j]$
$\qquad\quad \lor\ \neg grant \land \text{UNCHANGED}\ votedFor$
$\qquad \land\ Reply([mtype \mapsto RequestVoteResponse,$
$\qquad\quad mterm \mapsto currentTerm[i],$
$\qquad\quad mvoteGranted \mapsto grant,$
$\qquad\quad$ *mlog* is used just for the *elections* history variable for the proof. It would not exist in a real implementation.
$\qquad\quad mlog \mapsto log[i],$
$\qquad\quad msource \mapsto i,$
$\qquad\quad mdest \mapsto j],$
$\qquad\quad m)$
$\qquad \land\ \text{UNCHANGED}\ \langle state,\ currentTerm,\ candidateVars,\ leaderVars,\ logVars\rangle$

Server i receives a *RequestVote* response from server j with $m.mterm = currentTerm[i]$.

$HandleRequestVoteResponse(i,\ j,\ m) \triangleq$
This tallies votes even when the current state is not *Candidate*, but they won't be looked at, so it doesn't matter.
$\quad \land\ m.mterm = currentTerm[i]$
$\quad \land\ votesResponded' = [votesResponded\ \text{EXCEPT}\ ![i] =$

8

```
                                    votesResponded[i] ∪ {j}]
   ∧ ∨ ∧ m.mvoteGranted
         ∧ votesGranted′ = [votesGranted EXCEPT ![i] =
                                  votesGranted[i] ∪ {j}]
         ∧ voterLog′ = [voterLog EXCEPT ![i] =
                               voterLog[i] @@ (j :> m.mlog)]
     ∨ ∧ ¬m.mvoteGranted
         ∧ UNCHANGED ⟨votesGranted, voterLog⟩
   ∧ Discard(m)
   ∧ UNCHANGED ⟨serverVars, votedFor, leaderVars, logVars⟩
```

Server i receives an *AppendEntries* request from server j with $m.mterm \leq currentTerm[i]$. This just handles $m.entries$ of length 0 or 1, but implementations could safely accept more by treating them the same as multiple independent requests of 1 entry.

```
HandleAppendEntriesRequest(i, j, m) ≜
   LET logOk ≜ ∨ m.mprevLogIndex = 0
               ∨ ∧ m.mprevLogIndex > 0
                 ∧ m.mprevLogIndex ≤ Len(log[i])
                 ∧ m.mprevLogTerm = log[i][m.mprevLogIndex].term
   IN   ∧ m.mterm ≤ currentTerm[i]
        ∧ ∨ ∧  reject request
                ∨ m.mterm < currentTerm[i]
                ∨ ∧ m.mterm = currentTerm[i]
                  ∧ state[i] = Follower
                  ∧ ¬logOk
              ∧ Reply([mtype          ↦ AppendEntriesResponse,
                       mterm          ↦ currentTerm[i],
                       msuccess       ↦ FALSE,
                       mmatchIndex    ↦ 0,
                       msource        ↦ i,
                       mdest          ↦ j],
                       m)
              ∧ UNCHANGED ⟨serverVars, logVars⟩
          ∨  return to follower state
              ∧ m.mterm = currentTerm[i]
              ∧ state[i] = Candidate
              ∧ state′ = [state EXCEPT ![i] = Follower]
              ∧ UNCHANGED ⟨currentTerm, votedFor, logVars, messages⟩
          ∨  accept request
              ∧ m.mterm = currentTerm[i]
              ∧ state[i] = Follower
              ∧ logOk
              ∧ LET index ≜ m.mprevLogIndex + 1
                IN   ∨  already done with request
```

```
                 ∧ ∨ m.mentries = ⟨⟩
                   ∨ ∧ Len(log[i]) ≥ index
                     ∧ log[i][index].term = m.mentries[1].term
                  This could make our commitIndex decrease (for
                  example if we process an old, duplicated request),
                  but that doesn't really affect anything.
                 ∧ commitIndex' = [commitIndex EXCEPT ![i] =
                                        m.mcommitIndex]
                 ∧ Reply([mtype         ↦ AppendEntriesResponse,
                          mterm         ↦ currentTerm[i],
                          msuccess      ↦ TRUE,
                          mmatchIndex   ↦ m.mprevLogIndex +
                                            Len(m.mentries),
                          msource       ↦ i,
                          mdest         ↦ j],
                          m)
                 ∧ UNCHANGED ⟨votedFor, currentTerm, log, state⟩
               ∨  conflict: remove 1 entry
                 ∧ m.mentries ≠ ⟨⟩
                 ∧ Len(log[i]) ≥ index
                 ∧ log[i][index].term    ≠ m.mentries[1].term
                 ∧ LET new ≜ [index2 ∈ 1 . . (Len(log[i]) − 1) ↦
                                    log[i][index2]]
                   IN   log' = [log EXCEPT ![i] = new]
                 ∧ UNCHANGED ⟨serverVars, commitIndex, messages⟩
               ∨  no conflict: append entry
                 ∧ m.mentries ≠ ⟨⟩
                 ∧ Len(log[i]) = m.mprevLogIndex
                 ∧ log' = [log EXCEPT ![i] =
                              Append(log[i], m.mentries[1])]
                 ∧ UNCHANGED ⟨serverVars, commitIndex, messages⟩
          ∧ UNCHANGED ⟨candidateVars, leaderVars⟩

 Server i receives an AppendEntries response from server j with
 m.mterm = currentTerm[i].
HandleAppendEntriesResponse(i, j, m) ≜
    ∧ m.mterm = currentTerm[i]
    ∧ ∨ ∧ m.msuccess  successful
        ∧ nextIndex'  = [nextIndex  EXCEPT ![i][j]  = m.mmatchIndex + 1]
        ∧ matchIndex' = [matchIndex EXCEPT ![i][j] = m.mmatchIndex]
      ∨ ∧ ¬m.msuccess  not successful
        ∧ nextIndex' = [nextIndex EXCEPT ![i][j] =
                            Max({nextIndex[i][j] − 1, 1})]
        ∧ UNCHANGED ⟨matchIndex⟩
    ∧ Discard(m)
```

```
    ∧ UNCHANGED ⟨serverVars, candidateVars, logVars, elections⟩
```

Detached server i receives a *CatchupRequest* from leader j.

```
HandleCatchupRequest(i, j, m) ≜
  ∨ ∧ m.mterm < currentTerm[i]
    ∧ Reply([mtype ↦ CatchupResponse,
             mterm ↦ currentTerm[i],
             msuccess ↦ FALSE,
             mmatchIndex ↦ 0,
             msource ↦ i,
             mdest ↦ j,
             mroundsLeft ↦ 0],
             m)
    ∧ UNCHANGED ⟨serverVars, candidateVars,
         leaderVars, logVars⟩
  ∨ ∧ m.mterm ≥ currentTerm[i]
    ∧ currentTerm′ = [currentTerm EXCEPT ![i] = m.mterm]
    ∧ log′ = [log EXCEPT ![i] = SubSeq(log[i], 1, m.mlogLen) ∘ m.mentries]
    ∧ Reply([mtype ↦ CatchupResponse,
             mterm ↦ currentTerm[i],
             msuccess ↦ TRUE,
             mmatchIndex ↦ Len(log[i]),
             msource ↦ i,
             mdest ↦ j,
             mroundsLeft ↦ m.mrounds − 1],
            m)
    ∧ UNCHANGED ⟨state, votedFor, candidateVars, leaderVars,
         commitIndex⟩
```

Leader i receives a *CatchupResponse* from detached server j.

```
HandleCatchupResponse(i, j, m) ≜
```

A real system checks for progress every timeout interval.
Assume that if this response is called, the new server
has made progress.

```
  ∧ ∨ ∧ m.msuccess
      ∧ ∨ ∧ m.mmatchIndex ≠ commitIndex[i]
          ∧ m.mmatchIndex ≠ matchIndex[i][j]
        ∨ m.mmatchIndex = commitIndex[i]
      ∧ state[i] = Leader
      ∧ m.mterm = currentTerm[i]
      ∧ j ∉ GetConfig(i)
      ∧ nextIndex′ = [nextIndex EXCEPT ![i][j] = m.mmatchIndex + 1]
      ∧ matchIndex′ = [matchIndex EXCEPT ![i][j] = m.mmatchIndex]
      ∧ ∨ ∧ m.mroundsLeft ≠ 0
          ∧ Reply([mtype ↦ CatchupRequest,
```

```
                                    mterm ↦ currentTerm[i],
                                    mentries ↦ SubSeq(log[i],
                                                      nextIndex[i][j],
                                                      commitIndex[i]),
                                    mLogLen ↦ nextIndex[i][j] − 1,
                                    msource  ↦ i,
                                    mdest ↦ j,
                                    mrounds ↦ m.mroundsLeft],
                                    m)
                   ∨ ∧ m.mroundsLeft = 0
                       A real system makes sure the final call to this handler is
                       received after a timeout interval.
                       We assume that if a timeout happened, the message
                       has already been dropped.
                     ∧ Reply([mtype ↦ CheckOldConfig,
                              mterm ↦ currentTerm[i],
                              madd ↦ TRUE,
                              mserver ↦ j,
                              msource ↦ i,
                              mdest ↦ i], m)
               ∧ UNCHANGED ⟨elections⟩
            ∨ ∧ ∨ ¬m.msuccess
                 ∨ ∧ ∨ m.mmatchIndex = commitIndex[i]
                       ∨ m.mmatchIndex = matchIndex[i][j]
                   ∧ m.mmatchIndex ≠ commitIndex[i]
                 ∨ state[i] ≠ Leader
                 ∨ m.mterm ≠ currentTerm[i]
                 ∨ j ∈ GetConfig(i)
               ∧ Discard(m)
               ∧ UNCHANGED ⟨leaderVars⟩
        ∧ UNCHANGED ⟨serverVars, candidateVars, logVars⟩

 Leader i receives a CheckOldConfig message.
HandleCheckOldConfig(i, m) ≜
    ∨ ∧ state[i] ≠ Leader ∨ m.mterm = currentTerm[i]
      ∧ Discard(m)
      ∧ UNCHANGED ⟨serverVars, candidateVars, leaderVars, logVars⟩
    ∨ ∧ state[i] = Leader ∧ m.mterm = currentTerm[i]
      ∧ ∨ ∧ GetMaxConfigIndex(i) ≤ commitIndex[i]
          ∧ LET newConfig ≜ IF m.madd THEN UNION {GetConfig(i), {m.mserver}}
                                      ELSE  GetConfig(i) \ {m.mserver}
                newEntry ≜ [term ↦ currentTerm[i], type ↦ ConfigEntry, value ↦ newConfig]
                newLog ≜ Append(log[i], newEntry)
             IN    log′ = [log EXCEPT ![i] = newLog]
          ∧ Discard(m)
```

```
        ∧ UNCHANGED ⟨commitIndex⟩
     ∨ ∧ GetMaxConfigIndex(i) > commitIndex[i]
        ∧ Reply([mtype ↦ CheckOldConfig,
                 mterm ↦ currentTerm[i],
                 madd ↦ m.madd,
                 mserver ↦ m.mserver,
                 msource ↦ i,
                 mdest ↦ i],
                 m)
        ∧ UNCHANGED ⟨logVars⟩
   ∧ UNCHANGED ⟨serverVars, candidateVars, leaderVars⟩
```

Any *RPC* with a newer term causes the recipient to advance its term first.

```
UpdateTerm(i, j, m) ≜
   ∧ m.mterm > currentTerm[i]
   ∧ currentTerm' = [currentTerm EXCEPT ![i] = m.mterm]
   ∧ state'       = [state       EXCEPT ![i] = Follower]
   ∧ votedFor'    = [votedFor    EXCEPT ![i] = Nil]
      messages is unchanged so m can be processed further.
   ∧ UNCHANGED ⟨messages, candidateVars, leaderVars, logVars⟩
```

Responses with stale terms are ignored.

```
DropStaleResponse(i, j, m) ≜
   ∧ m.mterm < currentTerm[i]
   ∧ Discard(m)
   ∧ UNCHANGED ⟨serverVars, candidateVars, leaderVars, logVars⟩
```

Receive a message.

```
Receive(m) ≜
   LET i ≜ m.mdest
       j ≜ m.msource
   IN   Any RPC with a newer term causes the recipient to advance
        its term first. Responses with stale terms are ignored.
       ∨ UpdateTerm(i, j, m)
       ∨ ∧ m.mtype = RequestVoteRequest
         ∧ HandleRequestVoteRequest(i, j, m)
       ∨ ∧ m.mtype = RequestVoteResponse
         ∧ ∨ DropStaleResponse(i, j, m)
           ∨ HandleRequestVoteResponse(i, j, m)
       ∨ ∧ m.mtype = AppendEntriesRequest
         ∧ HandleAppendEntriesRequest(i, j, m)
       ∨ ∧ m.mtype = AppendEntriesResponse
         ∧ ∨ DropStaleResponse(i, j, m)
           ∨ HandleAppendEntriesResponse(i, j, m)
       ∨ ∧ m.mtype = CatchupRequest
         ∧ HandleCatchupRequest(i, j, m)
```

$\lor \land m.mtype = CatchupResponse$
$\quad \land HandleCatchupResponse(i, j, m)$
$\lor \land m.mtype = CheckOldConfig$
$\quad \land HandleCheckOldConfig(i, m)$

End of message handlers.

Network state transitions

The network duplicates a message

$DuplicateMessage(m) \triangleq$
$\quad \land Send(m)$
$\quad \land \text{UNCHANGED } \langle serverVars, candidateVars, leaderVars, logVars \rangle$

The network drops a message

$DropMessage(m) \triangleq$
$\quad \land Discard(m)$
$\quad \land \text{UNCHANGED } \langle serverVars, candidateVars, leaderVars, logVars \rangle$

Model invariants.

Safety property that only a single leader can be elected at a time.

$OneLeader \triangleq Cardinality(\{i \in Server : state[i] = Leader\}) \leq 1$

Defines how the variables may transition.

$Next \triangleq \land \lor \exists i \in Server : Restart(i)$
$\quad \lor \exists i \in Server : Timeout(i)$
$\quad \lor \exists i, j \in Server : RequestVote(i, j)$
$\quad \lor \exists i \in Server : BecomeLeader(i)$
$\quad \lor \exists i \in Server, v \in Value : ClientRequest(i, v)$
$\quad \lor \exists i, j \in Server : AddNewServer(i, j)$
$\quad \lor \exists i, j \in Server : DeleteServer(i, j)$
$\quad \lor \exists i \in Server : AdvanceCommitIndex(i)$
$\quad \lor \exists i, j \in Server : AppendEntries(i, j)$
$\quad \lor \exists m \in \text{DOMAIN } messages : Receive(m)$
$\quad \lor \exists m \in \text{DOMAIN } messages : DuplicateMessage(m)$
$\quad \lor \exists m \in \text{DOMAIN } messages : DropMessage(m)$
$\quad$ History variable that tracks every *log* ever:
$\quad \land allLogs' = allLogs \cup \{log[i] : i \in Server\}$

The specification must start with the initial state and transition according to *Next*.

$Spec \triangleq Init \land \Box[Next]_{vars}$

14

```
\* Changelog:
\*
\* 2015 - 05 - 10:
\* - Add cluster membership changes as described in Section 4 of
\*     Diego Ongaro. Consensus: Bridging theory and practice.
\*     PhD thesis, Stanford University, 2014.
\* This introduces: InitServer, ValueEntry, ConfigEntry, CatchupRequest,
\*     CatchupResponse, CheckOldConfig, GetMaxConfigIndex,
\*     GetConfig (parameterized), AddNewServer, DeleteServer,
\*     HandleCatchupRequest, HandleCatchupResponse,
\*     HandleCheckOldConfig
\*
\* 2014 - 12 - 02:
\* - Fix AppendEntries to only send one entry at a time, as originally
\* intended. Since SubSeq is inclusive, the upper bound of the range should
\* have been nextIndex, not nextIndex + 1. Thanks to Igor Kovalenko for
\* reporting the issue.
\* - Change matchIndex' to matchIndex (without the apostrophe) in
\* AdvanceCommitIndex. This apostrophe was not intentional and perhaps
\* confusing, though it makes no practical difference (matchIndex' equals
\* matchIndex). Thanks to Hugues Evrard for reporting the issue.
\*
\* 2014 - 07 - 06:
\* - Version from PhD dissertation
```