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.
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 a 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.
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.
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.**

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

---

### 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.**

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

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.

\[ \text{Quorum(config)} \triangleq \{ i \in \text{SUBSET (config)} : \text{Cardinality}(i) \times 2 > \text{Cardinality(config)} \} \]

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.

\[ \text{GetMaxConfigIndex}(i) \triangleq \]
\[ \text{LET} \quad \text{configIndexes} \triangleq \{ \text{index} \in 1 \ldots \text{Len(log}[i]) : \text{log}[i][\text{index}].\text{type} = \text{ConfigEntry} \} \]
\[ \text{IN} \quad \text{IF} \quad \text{configIndexes} = \{ \} \quad \text{THEN} \quad 0 \]
\[ \quad \text{ELSE} \quad \text{Max}(\text{configIndexes}) \]

Return the configuration of teh latest configuration in server \( i \)’s log.

\[ \text{GetConfig}(i) \triangleq \]
\[ \text{IF} \quad \text{GetMaxConfigIndex}(i) = 0 \quad \text{THEN} \quad \text{InitServer} \]
\[ \quad \text{ELSE} \quad \text{log}[i][\text{GetMaxConfigIndex}(i)].\text{value} \]
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 its 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 $j$.

$$\text{HandleCatchupRequest}(i, j, m) \triangleq$$

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

$$\land \text{Reply}([mtype \mapsto \text{CatchupResponse},$$

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

$$msuccess \mapsto \text{FALSE},$$

$$mmatchIndex \mapsto 0,$$

$$msource \mapsto i,$$

$$mdest \mapsto j,$$

$$mroundsLeft \mapsto 0],$$

$$m)$$

$$\land \text{UNCHANGED} (serverVars, candidateVars,$$

$$\land \text{leaderVars, logVars})$$

$$\lor \land m.mterm \geq currentTerm[i]$$

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

$$\land \text{log}' = [\text{log EXCEPT } ![i] = \text{SubSeq}(	ext{log}[i], 1, m.mlogLen) \circ m.\text{entries}]$$

$$\land \text{Reply}([mtype \mapsto \text{CatchupResponse},$$

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

$$msuccess \mapsto \text{TRUE},$$

$$mmatchIndex \mapsto \text{Len}(	ext{log}[i]),$$

$$msource \mapsto i,$$

$$mdest \mapsto j,$$

$$mroundsLeft \mapsto m.mrounds - 1],$$

$$m)$$

$$\land \text{UNCHANGED} (state, votedFor, candidateVars, leaderVars,$$

$$\land \text{commitIndex})$$

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 $\text{CatchupResponse}$ from detached server $j$.

\begin{align*}
\text{HandleCatchupResponse}(i, j, m) & \triangleq \\
\text{A real system checks for progress every timeout interval.} \\
\text{Assume that if this response is called, the new server has made progress.} \\
\& \lor \& m.\text{msuccess} \\
\& \lor \& \& m.\text{mmatchIndex} \neq \text{commitIndex}[i] \\
\& \lor \& \& m.\text{mmatchIndex} \neq \text{matchIndex}[i][j] \\
\lor \& m.\text{mmatchIndex} = \text{commitIndex}[i] \\
\& \text{state}[i] = \text{Leader} \\
\& \& m.\text{mterm} = \text{currentTerm}[i] \\
\& j \notin \text{GetConfig}(i) \\
\& \text{nextIndex}' = [\text{nextIndex} \text{ EXCEPT !}[i][j] = m.\text{mmatchIndex} + 1] \\
\& \text{matchIndex}' = [\text{matchIndex} \text{ EXCEPT !}[i][j] = m.\text{mmatchIndex}] \\
\& \lor \& \& m.\text{mroundsLeft} \neq 0 \\
\& \text{Reply}([m.\text{mtype} \mapsto \text{CatchupRequest},} \\
\& \text{mterm} \mapsto \text{currentTerm}[i], \\
\& \text{mentries} \mapsto \text{SubSeq}(\log[i],} \\
\& \text{nextIndex}[i][j], \\
\& \text{commitIndex}[i]), \\
\& \text{mLogLen} \mapsto \text{nextIndex}[i][j] - 1, \\
\& \text{msource} \mapsto i, \\
\& \text{mdest} \mapsto j, \\
\& \text{mrounds} \mapsto \text{m.mroundsLeft}, \\
\& m) \\
\lor \& \text{m.mroundsLeft} = 0 \\
\text{A real system makes sure the final call to this handler is received after a timeout interval.} \\
\text{We assume that if a timeout happened, the message has already been dropped.} \\
\& \text{Reply}([m.\text{mtype} \mapsto \text{CheckOldConfig},} \\
\& \text{mterm} \mapsto \text{currentTerm}[i], \\
\& \text{madd} \mapsto \text{TRUE}, \\
\& \text{mserver} \mapsto j, \\
\& \text{msource} \mapsto i, \\
\& \text{mdest} \mapsto i], m) \\
\lor \& \& \text{UNCHANGED } \langle \text{elections} \rangle \\
\lor \& \lor \neg m.\text{msuccess} \\
\& \lor \lor \& m.\text{mmatchIndex} = \text{commitIndex}[i] \\
\& \lor m.\text{mmatchIndex} = \text{matchIndex}[i][j] \\
\& m.\text{mmatchIndex} \neq \text{commitIndex}[i] \\
\lor \text{state}[i] \neq \text{Leader}
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.

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

Figure 3: An example of how a server can be disruptive even before the $C_{\text{new}}$ 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 $C_{\text{new}}$ entry in its log, but it hasn’t yet replicated that entry. Even before $C_{\text{new}}$ 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 $s \in \text{Quorum}(c_1)$, $t \in \text{Quorum}(c_2)$, then $s \cap t \neq \emptyset$. 
Proof.

\[ |s| \geq \left\lceil \frac{n}{2} \right\rceil + 1 \]
\[ |t| \geq \left\lceil \frac{n-1}{2} \right\rceil + 1 \]
\[ |s| + |t| \geq n + 1 \]

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 \( s \in \text{Quorum}(c_1), t \in \text{Quorum}(c_2) \), then \( s \cap t \neq \emptyset \).

Proof.

\[ |s| \geq \left\lceil \frac{n}{2} \right\rceil + 1 \]
\[ |t| \geq \left\lceil \frac{n+1}{2} \right\rceil + 1 \]
\[ |s| + |t| \geq n + 2 \]

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 \( s \in \text{Quorum}(c_1), t \in \text{Quorum}(c_2) \), then \( s \cap t \neq \emptyset \).

Proof.

\[ |s| \geq \left\lceil \frac{n-1}{2} \right\rceil + 1 \]
\[ |t| \geq \left\lceil \frac{n+1}{2} \right\rceil + 1 \]
\[ |s| + |t| \geq n + 2 \]

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 \( C_{\text{latest}} \) be the latest committed config and \( C_{\text{latest}-1} \) be the config that is committed right before \( C_{\text{latest}} \).

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 \( C_{\text{latest}} \) is committed, at least \( \left\lceil \frac{n}{2} \right\rceil + 1 \) servers have \( C_{\text{latest}} \) 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 \( \left\lceil \frac{n-1}{2} \right\rceil + 1 \) votes.

However, any server with \( C_{\text{latest}} \) 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 \( n - \left\lfloor \frac{n}{2} \right\rfloor - 1 \) 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 \( C_{\text{latest}} \) is committed, a quorum cannot be formed based on \( C_{\text{latest}} - 1 \). Induction can show that any config prior to \( C_{\text{latest}} \) cannot be the basis to form a quorum.

**Lemma 5.** Let \( C_{\text{latest}} \) be the latest committed config. Let \( C_{\text{new}} \) be any uncommitted config in the system, suppose \( C_{\text{latest}} = \{1, \ldots, n\} \). Then, \( C_{\text{new}} \) 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 \( C_{\text{latest}} \) in its log. Since in \texttt{HandleCheckOldConfig}, we require \( \text{GetMaxConfigIndex}(i) \leq \text{commitIndex}(i) \) to hold before the leader can append any newer config to its log, \( C_{\text{new}} \) can only be “one step” away from \( C_{\text{latest}} \).

**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].

\[
\forall e, f \in \text{elections} \\
\text{e.eterm} = f.eterm \Rightarrow \text{e.eleader} = f.eleader
\]

**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 \( C_{\text{latest}} \) and \( C_{\text{new}+} \) are possible.

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

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

\[
\text{grant} \triangleq m.\text{mterm} = \text{currentTerm}[i] \ (1) \\
\quad \land \text{logOk} \\
\quad \land \text{votedFor}[i] \in \{\text{Nil}, j\} \ (3)
\]

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

Since \( s \in e.evotes \) and \( s \in f.evotes \), \( e.eleader = f.eleader \).
• **Case 2.** $e.evotes, f.evotes \in \text{Quorum}(C_{new^+})$. Similar proof to **Case 1**.

• **Case 3.** $e.evotes, f.evotes \in \text{Quorum}(C_{new^-})$. Similar proof to **Case 1**.

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

• **Case 5.** $e.evotes \in \text{Quorum}(C_{\text{latest}}), f.evotes \in \text{Quorum}(C_{new^-})$.
  
  By Lemma 1, $e.evotes \cap f.evotes \neq \emptyset$. Afterwards, similar proof to **Case 1**.

• **Case 6.** $e.evotes \in \text{Quorum}(C_{new^+}), f.evotes \in \text{Quorum}(C_{new^-})$.
  
  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.

\[\square\]

### 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 its 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.

\[
\begin{align*}
s_1^* & \quad (1, \text{config, \{1,2,3\}}) \\
s_2 & \quad (1, \text{config, \{1,2,3\}}) \\
s_3 & \quad (1, \text{config, \{1,2,3\}}) \\
s_4 & \\
\end{align*}
\]
$s_1$ gets a request to add $s_4$, so catches up $s_4$ with the config entry.

\[
\begin{align*}
{s_1^*} & \quad (1, \text{config, \{1,2,3\}}) \\
{s_2} & \quad (1, \text{config, \{1,2,3\}}) \\
{s_3} & \quad (1, \text{config, \{1,2,3\}}) \\
{s_4} & \quad (1, \text{config, \{1,2,3\}}) \\
\end{align*}
\]

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

\[
\begin{align*}
{s_1^*} & \quad (1, \text{config, \{1,2,3\}}, (1, \text{config, \{1,2,3,4\}}) \\
{s_2} & \quad (1, \text{config, \{1,2,3\}}) \\
{s_3} & \quad (1, \text{config, \{1,2,3\}}) \\
{s_4} & \quad (1, \text{config, \{1,2,3\}}) \\
\end{align*}
\]

$s_3$ dies and $s_1$ replicates the new config to $s_2$.

\[
\begin{align*}
{s_1^*} & \quad (1, \text{config, \{1,2,3\}}, (1, \text{config, \{1,2,3,4\}}) \\
{s_2} & \quad (1, \text{config, \{1,2,3\}}, (1, \text{config, \{1,2,3,4\}}) \\
{s_3} & \quad (1, \text{config, \{1,2,3\}}) \\
{s_4} & \quad (1, \text{config, \{1,2,3\}}) \\
\end{align*}
\]

$s_2$ 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 $n$.

\[
\begin{align*}
{s_1^{*1}} & \quad (1, \text{config, \{1,2,3,4\}}) \\
{s_2} & \quad (1, \text{config, \{1,2,3,4\}}) \\
{s_3} & \quad (1, \text{config, \{1,2,3,4\}}) \\
{s_4} & \quad (1, \text{config, \{1,2,3,4\}}) \\
{s_5} & \quad \\
\end{align*}
\]

$s_1$ catches up $s_5$.

\[
\begin{align*}
{s_1^{*1}} & \quad (1, \text{config, \{1,2,3,4\}}) \\
{s_2} & \quad (1, \text{config, \{1,2,3,4\}}) \\
{s_3} & \quad (1, \text{config, \{1,2,3,4\}}) \\
{s_4} & \quad (1, \text{config, \{1,2,3,4\}}) \\
{s_5} & \quad (1, \text{config, \{1,2,3,4\}}) \\
\end{align*}
\]

$s_1$ appends new config.
\(s_1^{s_1}\) (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 \(s_5\).

\(s_1^{s_1}\) (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.

\(s_1^{D_{s_1}}\) (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.

\(s_1^{D_{s_1}}\) (1, config, \{1,2,3,4\}), (1,config,\{1,2,3,4,5\})
\(s_2^{T_{s_2}}\) (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\})

\(s_2, s_3, s_4\) vote for \(s_2\). \(s_5\) rejects. \(s_2\) becomes leader.

\(s_1^{D_{s_1}}\) (1, config, \{1,2,3,4\}), (1,config,\{1,2,3,4,5\})
\(s_2^{s_2}\) (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\})

\(s_2\) appends a new config to its log.

\(s_1^{D_{s_1}}\) (1, config, \{1,2,3,4\}), (1,config,\{1,2,3,4,5\})
\(s_2^{s_2}\) (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!

17
\(s_1^{D*1}\) (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


A Original TLA+ Specification

Starts on next page.
MODULE raft_orig

This is the formal specification for the Raft consensus algorithm.

Copyright 2014 Diego Ongaro.
This work is licensed under the Creative Commons Attribution – 4.0
International License https://creativecommons.org/licenses/by/4.0/

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`

\[\text{serverVars} \triangleq \langle \text{currentTerm, state, votedFor} \rangle\]

A Sequence of log entries. The index into this sequence is the index of the log entry. Unfortunately, the Sequence module defines \text{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`

\[\text{logVars} \triangleq \langle \text{log, commitIndex} \rangle\]

The following variables are used only on candidates:

- The set of servers from which the candidate has received a \text{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`

\[\text{candidateVars} \triangleq \langle \text{votesResponded, votesGranted, voterLog} \rangle\]

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 \text{commitIndex} on the leader.

VARIABLE `matchIndex`

\[\text{leaderVars} \triangleq \langle \text{nextIndex, matchIndex, elections} \rangle\]

End of per server variables.

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

\[\text{vars} \triangleq \langle \text{messages, allLogs, serverVars, candidateVars, leaderVars, logVars} \rangle\]

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

\[
\text{Quorum} \triangleq \{i \in \text{SUBSET (Server)} : \text{Cardinality}(i) \ast 2 > \text{Cardinality}(\text{Server})\}
\]

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

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

Helper for \textit{Send} and \textit{Reply}. Given a message \(m\) and bag of messages, return a new bag of messages with one more \(m\) in it.

\[
\text{WithMessage}(m, \text{msgs}) \triangleq
\begin{cases}
    \text{IF } m \in \text{DOMAIN } \text{msgs} \text{ THEN } & \text{msgs EXCEPT } ![m] = \text{msgs}[m] + 1 \\
    \text{ELSE } & \text{msgs } \ominus \ominus (m :> 1)
\end{cases}
\]

Helper for \textit{Discard} and \textit{Reply}. Given a message \(m\) and bag of messages, return a new bag of messages with one less \(m\) in it.

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

Add a message to the bag of messages.

\[
\text{Send}(m) \triangleq \text{messages'} = \text{WithMessage}(m, \text{messages})
\]

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

\[
\text{Discard}(m) \triangleq \text{messages'} = \text{WithoutMessage}(m, \text{messages})
\]

Combination of \textit{Send} and \textit{Discard}

\[
\text{Reply}(\text{response}, \text{request}) \triangleq \text{messages'} = \text{WithoutMessage(request, WithMessage(response, messages))}
\]

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

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

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

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

Define initial values for all variables

\[
\text{InitHistoryVars} \triangleq \land \text{elections} = \{\} \\
\land \text{allLogs} = \{\} \\
\land \text{voterLog} = [i \in \text{Server} \mapsto [j \in \{\} \mapsto \langle\}]]
\]

\[
\text{InitServerVars} \triangleq \land \text{currentTerm} = [i \in \text{Server} \mapsto 1] \\
\land \text{state} = [i \in \text{Server} \mapsto \text{Follower}] \\
\land \text{votedFor} = [i \in \text{Server} \mapsto \text{Nil}]
\]
\[ \text{InitCandidateVars} \triangleq \land \text{votesResponded} = \left[ i \in \text{Server} \mapsto \{ \} \right] \land \text{votesGranted} = \left[ i \in \text{Server} \mapsto \{ \} \right] \]

The values \( \text{nextIndex}[i][i] \) and \( \text{matchIndex}[i][i] \) are never read, since the leader does not send itself messages. It’s still easier to include these in the functions.

\[ \text{InitLeaderVars} \triangleq \land \text{nextIndex} = \left[ i \in \text{Server} \mapsto [j \in \text{Server} \mapsto 1] \right] \land \text{matchIndex} = \left[ i \in \text{Server} \mapsto [j \in \text{Server} \mapsto 0] \right] \]

\[ \text{InitLogVars} \triangleq \land \text{log} = \left[ i \in \text{Server} \mapsto \langle \rangle \right] \land \text{commitIndex} = \left[ i \in \text{Server} \mapsto 0 \right] \]

\[ \text{Init} \triangleq \land \text{messages} = \left[ m \in \{ \} \mapsto 0 \right] \land \text{InitHistoryVars} \land \text{InitServerVars} \land \text{InitCandidateVars} \land \text{InitLeaderVars} \land \text{InitLogVars} \]

Define state transitions

Server \( i \) restarts from stable storage.

It loses everything but its \text{currentTerm}, \text{votedFor}, and \text{log}.

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

Server \( i \) times out and starts a new election.

\[ \text{Timeout}(i) \triangleq \land \text{state}[i] \in \{ \text{Follower, Candidate} \} \land \text{state'} = \left[ \text{state \ except \ !}[i] = \text{Candidate} \right] \land \text{currentTerm'} = \left[ \text{currentTerm \ except \ !}[i] = \text{currentTerm}[i] + 1 \right] \]

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

\[ \land \text{votedFor'} = \left[ \text{votedFor \ except \ !}[i] = \text{Nil} \right] \land \text{votesResponded'} = \left[ \text{votesResponded \ except \ !}[i] = \{ \} \right] \land \text{votesGranted'} = \left[ \text{votesGranted \ except \ !}[i] = \{ \} \right] \land \text{voterLog'} = \left[ \text{voterLog \ except \ !}[i] = [j \in \{ \} \mapsto \langle \rangle ] \right] \land \text{UNCHANGED} \langle \text{messages, leaderVars, logVars} \rangle \]

Candidate \( i \) sends \( j \) a \text{RequestVote} request.

\[ \text{RequestVote}(i, j) \triangleq \land \text{state}[i] = \text{Candidate} \]

4
\[ j \notin \text{votesResponded}[i] \]
\[ \land \text{Send}([\text{mtype} \mapsto \text{RequestVoteRequest}, \text{mterm} \mapsto \text{currentTerm}[i], \text{mlastLogTerm} \mapsto \text{LastTerm}(\log[i]), \text{mlastLogIndex} \mapsto \text{Len}(\log[i]), \text{msource} \mapsto i, \text{mdest} \mapsto j]) \]
\[ \land \text{UNCHANGED } \langle \text{serverVars}, \text{candidateVars}, \text{leaderVars}, \logVars \rangle \]

Leader \( i \) sends \( j \) an \text{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.

\text{AppendEntries}(i, j) \triangleq \land i \neq j \
\land \text{state}[i] = \text{Leader} \\
\land \text{LET} \text{prevLogIndex} \triangleq \text{nextIndex}[i][j] - 1 \\
\text{prevLogTerm} \triangleq \begin{cases} \text{IF prevLogIndex} > 0 \text{ THEN} \\
\text{log}[i][\text{prevLogIndex}].\text{term} \\
\text{ELSE} \\
0 \end{cases} \\
\text{Send up to 1 entry, constrained by the end of the log.} \\
\text{lastEntry} \triangleq \text{Min}(\{\text{Len}(\log[i]), \text{nextIndex}[i][j]\}) \\
\text{entries} \triangleq \text{SubSeq}(\log[i], \text{nextIndex}[i][j], \text{lastEntry}) \\
\text{IN} \text{ Send}([\text{mtype} \mapsto \text{AppendEntriesRequest}, \text{mterm} \mapsto \text{currentTerm}[i], \text{mprevLogIndex} \mapsto \text{prevLogIndex}, \text{mprevLogTerm} \mapsto \text{prevLogTerm}, \text{mentries} \mapsto \text{entries}, \text{mlog} \text{ is used as a history variable for the proof.} \\
\text{It would not exist in a real implementation.} \\
\text{mlog} \mapsto \log[i], \\
\text{mcommitIndex} \mapsto \text{Min}(\{\text{commitIndex}[i], \text{lastEntry}\}), \\
\text{msource} \mapsto i, \\
\text{mdest} \mapsto j) \\
\land \text{UNCHANGED } \langle \text{serverVars}, \text{candidateVars}, \text{leaderVars}, \logVars \rangle \]

Candidate \( i \) transitions to leader.

\text{BecomeLeader}(i) \triangleq \land \text{state}[i] = \text{Candidate} \\
\land \text{votesGranted}[i] \in \text{Quorum} \\
\land \text{state}' = [\text{state EXCEPT ![i] = Leader}] \\
\land \text{nextIndex}' = [\text{nextIndex EXCEPT ![i] =} \\
[j \in \text{Server} \mapsto \text{Len}(\log[i]) + 1)] \\
\land \text{matchIndex}' = [\text{matchIndex EXCEPT ![i] =} \\
[j \in \text{Server} \mapsto 0]]
∧ elections’ = elections ∪
{[eterm ↦→ currentTerm[i],
leader ↦→ 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
\( m.mterm \leq currentTerm[i] \).

\[ \text{HandleRequestVoteRequest}(i, j, m) \triangleq \]

\[
\text{LET } \text{logOk} \triangleq \lor m.mlastLogTerm > \text{LastTerm}(log[i]) \]
\[
\quad \land m.mlastLogTerm = \text{LastTerm}(log[i]) \]
\[
\quad \land m.mlastLogIndex \geq \text{Len}(log[i]) \]
\[
\text{grant} \triangleq \land m.mterm = currentTerm[i] \]
\[
\quad \land \text{logOk} \]
\[
\quad \land \text{votedFor}[i] \in \{\text{Nil}, j\} \]
\[
\text{IN } \land \land \text{grant} \land \text{votedFor}' = [\text{votedFor EXCEPT } ![i] = j] \]
\[
\lor \neg \text{grant} \land \text{UNCHANGED} \text{ votedFor} \]
\[
\land \text{Reply}([\text{mtype} \mapsto \text{RequestVoteResponse}, \]
\[
\quad \text{mterm} \mapsto \text{currentTerm}[i], \]
\[
\quad \text{mvoteGranted} \mapsto \text{grant}, \]
\[
\quad \text{mlog} \mapsto \text{log}[i], \]
\[
\quad \text{msource} \mapsto i, \]
\[
\quad \text{mdest} \mapsto j, \]
\[
\quad m) \]
\[
\land \text{UNCHANGED} \langle \text{state}, \text{currentTerm}, \text{candidateVars}, \text{leaderVars}, \text{logVars} \rangle \]

Server \( i \) receives a \text{RequestVote} response from server \( j \) with \( m.mterm = \text{currentTerm}[i] \).

\[ \text{HandleRequestVoteResponse}(i, j, m) \triangleq \]

This tallies votes even when the current state is not \text{Candidate}, but they won’t be looked at, so it doesn’t matter.

\[
\land \land \text{grant} \land \text{votesResponded}' = [\text{votesResponded EXCEPT } ![i] = \text{votesResponded}[i] \cup \{j\}] \]
\[
\land \lor \land \text{mvoteGranted} \]
\[
\land \text{votesGranted}' = [\text{votesGranted EXCEPT } ![i] = \text{votesGranted}[i] \cup \{j\}] \]
\[
\land \text{voterLog}' = [\text{voterLog EXCEPT } ![i] = \text{voterLog}[i] @@ (j : m.mlog)] \]
\[
\lor \land \neg \text{mvoteGranted} \]
\[
\land \text{UNCHANGED} \langle \text{votesGranted}, \text{voterLog} \rangle \]
\[
\land \text{Discard}(m) \]
\[
\land \text{UNCHANGED} \langle \text{serverVars}, \text{votedFor}, \text{leaderVars}, \text{logVars} \rangle \]

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

\[ \text{HandleAppendEntriesRequest}(i, j, m) \triangleq \]
LET $logOk \triangleq \begin{cases} \lor m.mprevLogIndex = 0 \\ \land m.mprevLogIndex > 0 \\ \land m.mprevLogIndex \leq \text{Len}(log[i]) \\ \land m.mprevLogTerm = log[i][m.mprevLogIndex].\text{term} \end{cases}$

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

$\land \lor \land \text{reject request}$

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

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

$\land \land \text{state}[i] = \text{Follower}$

$\land \neg \text{logOk}$

$\land \text{Reply}([mtype \mapsto \text{AppendEntriesResponse},$

$mterm \mapsto currentTerm[i],$

$msuccess \mapsto \text{FALSE},$

$mmatchIndex \mapsto 0,$

$msource \mapsto i,$

$mdest \mapsto j],$

$m)$

$\land \text{UNCHANGED} \langle \text{serverVars, logVars} \rangle$

$\lor \text{return to follower state}$

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

$\land \land \text{state}[i] = \text{Candidate}$

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

$\land \text{UNCHANGED} \langle \text{currentTerm, votedFor, logVars, messages} \rangle$

$\lor \text{accept request}$

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

$\land \land \text{state}[i] = \text{Follower}$

$\land \land \text{logOk}$

$\land \text{LET} \ \text{index} \triangleq m.mprevLogIndex + 1$

IN $\lor \land \text{already done with request}$

$\land \lor \land \text{m.entries} = ()$

$\lor \land \text{Len}(log[i]) \geq \text{index}$

$\land \land log[i][\text{index}].\text{term} = \text{m.entries}[1].\text{term}$

This could make our $\text{commitIndex}$ decrease (for example if we process an old, duplicated request), but that doesn’t really affect anything.

$\land \text{commitIndex}' = [\text{commitIndex EXCEPT } ![i] = m.mcommitIndex]$

$\land \text{Reply}([mtype \mapsto \text{AppendEntriesResponse},$

$mterm \mapsto currentTerm[i],$

$msuccess \mapsto \text{TRUE},$

$mmatchIndex \mapsto m.mprevLogIndex + \text{Len}(m.mentries),$

$msource \mapsto i,$

$mdest \mapsto j],$

$m)$
∧ UNCHANGED ⟨serverVars, logVars⟩

∨ 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⟩

\( \text{Server } i \text{ receives an AppendEntries response from server } j \text{ with} \)
\( m.mterm = \text{currentTerm}[i]. \)
\( \text{HandleAppendEntriesResponse}(i, j, m) \triangleq \)
∧ m.mterm = currentTerm[i]
∧ \( \wedge \) m.msucceed successful
∧ nextIndex′ = [nextIndex EXCEPT ![i][j] = m.mmatchIndex + 1]
∧ matchIndex′ = [matchIndex EXCEPT ![i][j] = m.mmatchIndex]
∧ \( \wedge \) ¬m.msucceed not successful
∧ nextIndex′ = [nextIndex EXCEPT ![i][j] = Max(\{nextIndex[i][j] − 1, 1\})]
∧ UNCHANGED ⟨matchIndex⟩
∧ Discard(m)
∧ UNCHANGED ⟨serverVars, candidateVars, logVars, elections⟩

\( \text{Any RPC with a newer term causes the recipient to advance its term first.} \)
\( \text{UpdateTerm}(i, j, m) \triangleq \)
∧ 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⟩

\( \text{Responses with stale terms are ignored.} \)
\( \text{DropStaleResponse}(i, j, m) \triangleq \)
∧ m.mterm < currentTerm[i]
∧ Discard(m)
∧ UNCHANGED ⟨serverVars, candidateVars, leaderVars, logVars⟩
Receive a message.  
\[ \text{Receive}(m) \triangleq \]

\[
\begin{align*}
\text{LET } & \quad i \triangleq m.mdest \\
& \quad j \triangleq m.msource \\
\text{IN } & \quad \text{Any RPC with a newer term causes the recipient to advance its term first. Responses with stale terms are ignored.} \\
& \quad \lor \text{UpdateTerm}(i, j, m) \\
& \quad \lor m.mtype = \text{RequestVoteRequest} \\
& \quad \land \text{HandleRequestVoteRequest}(i, j, m) \\
& \quad \lor m.mtype = \text{RequestVoteResponse} \\
& \quad \land \lor \text{DropStaleResponse}(i, j, m) \\
& \quad \lor \text{HandleRequestVoteResponse}(i, j, m) \\
& \quad \lor m.mtype = \text{AppendEntriesRequest} \\
& \quad \land \lor \text{HandleAppendEntriesRequest}(i, j, m) \\
& \quad \lor m.mtype = \text{AppendEntriesResponse} \\
& \quad \land \lor \text{DropStaleResponse}(i, j, m) \\
& \quad \lor \text{HandleAppendEntriesResponse}(i, j, m)
\end{align*}
\]

End of message handlers.

Network state transitions

The network duplicates a message
\[ \text{DuplicateMessage}(m) \triangleq \]

\[
\begin{align*}
& \quad \land \text{Send}(m) \\
& \quad \land \text{UNCHANGED } \langle \text{serverVars, candidateVars, leaderVars, logVars} \rangle
\end{align*}
\]

The network drops a message
\[ \text{DropMessage}(m) \triangleq \]

\[
\begin{align*}
& \quad \land \text{Discard}(m) \\
& \quad \land \text{UNCHANGED } \langle \text{serverVars, candidateVars, leaderVars, logVars} \rangle
\end{align*}
\]

Defines how the variables may transition.
\[ \text{Next} \triangleq \]

\[
\begin{align*}
& \quad \land \lor \exists i \in \text{Server} : \text{Restart}(i) \\
& \quad \lor \exists i \in \text{Server} : \text{Timeout}(i) \\
& \quad \lor \exists i, j \in \text{Server} : \text{RequestVote}(i, j) \\
& \quad \lor \exists i \in \text{Server} : \text{BecomeLeader}(i) \\
& \quad \lor \exists i \in \text{Server}, v \in \text{Value} : \text{ClientRequest}(i, v) \\
& \quad \lor \exists i \in \text{Server} : \text{AdvanceCommitIndex}(i) \\
& \quad \lor \exists i, j \in \text{Server} : \text{AppendEntries}(i, j) \\
& \quad \lor \exists m \in \text{DOMAIN messages} : \text{Receive}(m) \\
& \quad \lor \exists m \in \text{DOMAIN messages} : \text{DuplicateMessage}(m) \\
& \quad \lor \exists m \in \text{DOMAIN messages} : \text{DropMessage}(m)
\end{align*}
\]

History variable that tracks every log ever:
\[ \land \text{allLogs}' = \text{allLogs} \cup \{ \text{log}[i] : i \in \text{Server} \} \]
The specification must start with the initial state and transition according to $\text{Next}$.

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

---

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

Starts on next page.
This is the formal specification for the Raft consensus algorithm.

Original Copyright 2014 Diego Ongaro
Modifications for cluster membership changes by
Brandon Amos and Huanchen Zhang, 2015

This work is licensed under the Creative Commons Attribution – 4.0
International License https://creativecommons.org/licenses/by/4.0/

EXTENDS Naturals, FiniteSets, Sequences, TLC

The number of rounds to catch new servers up for.
Must be $\geq 1$.

CONSTANTS $\text{NumRounds}$

The initial and global set of servers.
CONSTANTS $\text{InitServer}$, $\text{Server}$

Log metadata to distinguish values from configuration changes.
CONSTANT $\text{ValueEntry}$, $\text{ConfigEntry}$

The set of values that can go into the log.
CONSTANTS $\text{Value}$

Server states.
CONSTANTS $\text{Follower}$, $\text{Candidate}$, $\text{Leader}$

A reserved value.
CONSTANTS $\text{Nil}$

Message types:
CONSTANTS $\text{RequestVoteRequest}$, $\text{RequestVoteResponse}$,
$\text{AppendEntriesRequest}$, $\text{AppendEntriesResponse}$,
$\text{CatchupRequest}$, $\text{CatchupResponse}$,
$\text{CheckOldConfig}$

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

\[ serverVars \triangleq (currentTerm, state, votedFor) \]

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

\[ logVars \triangleq (log, commitIndex) \]

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

\[ candidateVars \triangleq (votesResponded, votesGranted, voterLog) \]

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

\[ \text{leaderVars} \triangleq (\text{nextIndex}, \text{matchIndex}, \text{elections}) \]

End of per server variables.

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

\[ \text{vars} \triangleq (\text{messages, allLogs, serverVars, candidateVars, leaderVars, logVars}) \]

**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.

\[ \text{Quorum}(\text{config}) \triangleq \{ i \in \text{SUBSET}(\text{config}) : \text{Cardinality}(i) \ast 2 > \text{Cardinality}(\text{config}) \} \]

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

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

Helper for \textit{Send} and \textit{Reply}. Given a message \( m \) and bag of messages, return a new bag of messages with one more \( m \) in it.

\[ \text{WithMessage}(m, \text{msgs}) \triangleq \]

\[ \text{IF } m \in \text{DOMAIN } \text{msgs} \text{ THEN} \]

\[ [\text{msgs EXCEPT } ![m] = \text{msgs}[m] + 1] \]

\[ \text{ELSE} \]

\[ \text{msgs} @@ (m :> 1) \]

Helper for \textit{Discard} and \textit{Reply}. Given a message \( m \) and bag of messages, return a new bag of messages with one less \( m \) in it.

\[ \text{WithoutMessage}(m, \text{msgs}) \triangleq \]

\[ \text{IF } m \in \text{DOMAIN } \text{msgs} \text{ THEN} \]

\[ [\text{msgs EXCEPT } ![m] = \text{msgs}[m] - 1] \]

\[ \text{ELSE} \]

\[ \text{msgs} \]

Add a message to the bag of messages.

\[ \text{Send}(m) \triangleq \text{messages'} = \text{WithMessage}(m, \text{messages}) \]

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

\[ \text{Discard}(m) \triangleq \text{messages'} = \text{WithoutMessage}(m, \text{messages}) \]

Combination of \textit{Send} and \textit{Discard}

\[ \text{Reply}(\text{response, request}) \triangleq \]
messages' = WithoutMessage(request, WithMessage(response, messages))

Return the minimum value from a set, or undefined if the set is empty.
\[ \text{Min}(s) \triangleq \text{CHOOSE } x \in s : \forall y \in s : x \leq y \]

Return the maximum value from a set, or undefined if the set is empty.
\[ \text{Max}(s) \triangleq \text{CHOOSE } x \in s : \forall y \in s : x \geq y \]

Return the index of the latest configuration in server i’s log.
\[ \text{GetMaxConfigIndex}(i) \triangleq \]
\[ \text{LET } \text{configIndexes} \triangleq \{ \text{index} \in 1..\text{Len}(\text{log}[i]) : \text{log}[i][\text{index}].\text{type} = \text{ConfigEntry} \} \]
\[ \text{IN } \text{IF } \text{configIndexes} = \{ \} \text{ THEN } 0 \]
\[ \text{ELSE } \text{Max}(\text{configIndexes}) \]

Return the configuration of the latest configuration in server i’s log.
\[ \text{GetConfig}(i) \triangleq \]
\[ \text{IF } \text{GetMaxConfigIndex}(i) = 0 \text{ THEN } \text{InitServer} \]
\[ \text{ELSE } \text{log}[i][\text{GetMaxConfigIndex}(i)].\text{value} \]

Define initial values for all variables

\[ \text{InitHistoryVars} \triangleq \]
\[ \land \text{elections} = \{ \} \]
\[ \land \text{allLogs} = \{ \} \]
\[ \land \text{voterLog} = [i \in \text{Server} \mapsto [j \in \{ \} \mapsto \{\}]] \]

\[ \text{InitServerVars} \triangleq \]
\[ \land \text{currentTerm} = [i \in \text{Server} \mapsto 1] \]
\[ \land \text{state} = [i \in \text{Server} \mapsto \text{Follower}] \]
\[ \land \text{votedFor} = [i \in \text{Server} \mapsto \text{Nil}] \]

\[ \text{InitCandidateVars} \triangleq \]
\[ \land \text{votesResponded} = [i \in \text{Server} \mapsto \{\}] \]
\[ \land \text{votesGranted} = [i \in \text{Server} \mapsto \{\}] \]

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.

\[ \text{InitLeaderVars} \triangleq \]
\[ \land \text{nextIndex} = [i \in \text{Server} \mapsto [j \in \text{Server} \mapsto 1]] \]
\[ \land \text{matchIndex} = [i \in \text{Server} \mapsto [j \in \text{Server} \mapsto 0]] \]

\[ \text{InitLogVars} \triangleq \]
\[ \land \text{log} = [i \in \text{Server} \mapsto \{\}] \]
\[ \land \text{commitIndex} = [i \in \text{Server} \mapsto 0] \]

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

Define state transitions
Server \(i\) restarts from stable storage.
It loses everything but its currentTerm, votedFor, and log.
\[ \text{Restart}(i) \triangleq \]
\[ \wedge i \in \text{GetConfig}(i) \]
\[ \wedge \text{state}' = [\text{state EXCEPT } ![i] = \text{Follower}] \]
\[ \wedge \text{votedFor}' = [\text{votedFor EXCEPT } ![i] = \text{Nil}] \]
\[ \wedge \text{votesResponded}' = [\text{votesResponded EXCEPT } ![i] = \{\}] \]
\[ \wedge \text{votesGranted}' = [\text{votesGranted EXCEPT } ![i] = \{\}] \]
\[ \wedge \text{nextLog}' = [\text{nextLog EXCEPT } ![i] = \text{LastTerm}([\text{log}[i]])] \]
\[ \wedge \text{matchIndex}' = [\text{matchIndex EXCEPT } ![i] = \text{Len}([\text{log}[i]])] \]
\[ \wedge \text{commitIndex}' = [\text{commitIndex EXCEPT } ![i] = 0] \]
\[ \wedge \text{voterLog}' = [\text{voterLog EXCEPT } ![i] = \{\}] \]
\[ \text{timeout}(i) \triangleq \]
\[ \text{timeout}(i) \triangleq \]
\[ \wedge \text{state}[i] \in \{\text{Follower}, \text{Candidate}\} \]
\[ \wedge \text{state}' = [\text{state EXCEPT } ![i] = \text{Candidate}] \]
\[ \wedge \text{currentTerm}' = [\text{currentTerm EXCEPT } ![i] = \text{currentTerm}[i] + 1] \]
\[ \text{Most implementations would probably just set the local vote atomically, but messaging localhost for it is weaker.} \]
\[ \wedge \text{votedFor}' = [\text{votedFor EXCEPT } ![i] = \text{Nil}] \]
\[ \wedge \text{votesResponded}' = [\text{votesResponded EXCEPT } ![i] = \{\}] \]
\[ \wedge \text{votesGranted}' = [\text{votesGranted EXCEPT } ![i] = \{\}] \]
\[ \wedge \text{voterLog}' = [\text{voterLog EXCEPT } ![i] = \{\}] \]
\[ \wedge \text{UNCHANGED } \langle \text{messages, currentTerm, votedFor, log, elections} \rangle \]

Client \(i\) times out and starts a new election.
\[ \text{RequestVote}(i, j) \triangleq \]
\[ \wedge j \in \text{GetConfig}(i) \setminus \text{votesResponded}[i] \]
\[ \wedge \text{Send}(\{\text{mtype} \mapsto \text{RequestVoteRequest}, \text{mterm} \mapsto \text{currentTerm}[i], \text{mlastLogTerm} \mapsto \text{LastTerm}([\text{log}[i]]), \text{mlastLogIndex} \mapsto \text{Len}([\text{log}[i]]), \text{msource} \mapsto i, \text{mdest} \mapsto j\}) \]
\[ \wedge \text{UNCHANGED } \langle \text{serverVars, candidateVars, leaderVars, logVars} \rangle \]

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.
\[ \text{AppendEntries}(i, j) \triangleq \]
\[ \wedge i \neq j \]
\[ \wedge \text{state}[i] = \text{Leader} \]
\[ \wedge j \in \text{GetConfig}(i) \]
\[ \wedge \text{LET prevLogIndex } \triangleq \text{nextIndex}[i][j] - 1 \]
prevLogTerm \overset{\Delta}{=} \begin{cases} \text{if } \text{prevLogIndex} > 0 \text{ then} \\
\log[i][\text{prevLogIndex}].\text{term} \\
\text{else} \\
0 \end{cases}

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

\begin{align*}
\text{lastEntry} & \overset{\Delta}{=} \text{Min}\left(\{\text{Len}(\log[i]), \text{nextIndex}[i][j]\}\right) \\
\text{entries} & \overset{\Delta}{=} \text{SubSeq}(\log[i], \text{nextIndex}[i][j], \text{lastEntry})
\end{align*}

\text{IN} \quad \text{Send}(\langle \text{mtype} \mapsto \text{AppendEntriesRequest}, \\
\text{mterm} \mapsto \text{currentTerm}[i], \\
\text{mprevLogIndex} \mapsto \text{prevLogIndex}, \\
\text{mprevLogTerm} \mapsto \text{prevLogTerm}, \\
\text{mentries} \mapsto \text{entries}, \\
\text{mlog} \text{ is used as a history variable for the proof.} \text{ It would not exist in a real implementation.} \\
\text{mlog} \mapsto \log[i], \\
\text{mcommitIndex} \mapsto \text{Min}\left(\{\text{commitIndex}[i], \text{lastEntry}\}\right) \\
\text{msource} \mapsto i, \\
\text{mdest} \mapsto j \rangle)

\wedge \text{UNCHANGED } \langle \text{serverVars, candidateVars, leaderVars, logVars} \rangle

\text{Candidate } i \text{ transitions to leader.}

\text{BecomeLeader}(i) \overset{\Delta}{=} 
\begin{align*}
\wedge \text{state}[i] & = \text{Candidate} \\
\wedge \text{votesGranted}[i] & \in \text{Quorum}(\text{GetConfig}(i)) \\
\wedge \text{state}' & = [\text{state} \text{ EXCEPT } ![i] = \text{Leader}] \\
\wedge \text{nextIndex}' & = [\text{nextIndex} \text{ EXCEPT } ![i] = \\
& [j \in \text{Server} \mapsto \text{Len}(\log[i]) + 1]] \\
\wedge \text{matchIndex}' & = [\text{matchIndex} \text{ EXCEPT } ![i] = \\
& [j \in \text{Server} \mapsto 0]] \\
\wedge \text{elections}' & = \text{elections} \cup \\
& \{[\text{eterm} \mapsto \text{currentTerm}[i], \\
\text{eleader} \mapsto i, \\
\text{elog} \mapsto \log[i], \\
\text{evotes} \mapsto \text{votesGranted}[i], \\
\text{evoterLog} \mapsto \text{voterLog}[i]\\]} \\
\wedge \text{UNCHANGED } \langle \text{messages, currentTerm, votedFor, candidateVars, logVars} \rangle
\end{align*}

\text{Leader } i \text{ receives a client request to add } v \text{ to the log.}

\text{ClientRequest}(i, v) \overset{\Delta}{=} 
\begin{align*}
\wedge \text{state}[i] & = \text{Leader} \\
\wedge \text{LET } \text{entry} & \overset{\Delta}{=} [\text{term} \mapsto \text{currentTerm}[i], \\
& \text{type} \mapsto \text{ValueEntry}, \\
& \text{value} \mapsto v] \\
\text{newLog} & \overset{\Delta}{=} \text{Append}(\log[i], \text{entry})
\end{align*}

\text{IN} \quad \log' = [\log \text{ EXCEPT } ![i] = \text{newLog}]

6

37
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.

$\text{AdvanceCommitIndex}(i) \triangleq$

\[
\begin{align*}
\wedge \text{state}[i] &= \text{Leader} \\
\wedge \text{LET } &\quad \text{The set of servers that agree up through index.} \\
\quad \text{Agree}(\text{index}) &\triangleq \{i\} \cup \{k \in \text{GetConfig}(i) : \\
\quad &\quad \text{matchIndex}[i][k] \geq \text{index}\} \\
\quad \text{The maximum indexes for which a quorum agrees} \\
\quad \text{agreeIndexes} &\triangleq \{\text{index} \in 1..\text{Len}(\text{log}[i]) : \\
\quad &\quad \text{Agree}(\text{index}) \in \text{Quorum}(\text{GetConfig}(i))\}
\end{align*}
\]

\[
\begin{align*}
\text{newCommitIndex} &\triangleq \\
\text{IF } &\quad \wedge \text{agreeIndexes} \neq \{\} \\
\quad \wedge \text{log}[i][\text{Max}(\text{agreeIndexes})].\text{term} = \text{currentTerm}[i] \\
\text{THEN } &\quad \text{Max}(\text{agreeIndexes}) \\
\text{ELSE } &\quad \text{commitIndex}[i] \\
\text{IN } &\quad \text{commitIndex}' = \text{commitIndex EXCEPT ![i] = newCommitIndex} \\
\wedge \text{UNCHANGED } \langle \text{messages, serverVars, candidateVars, leaderVars, commitIndex} \rangle
\end{align*}
\]

Leader $i$ adds a new server $j$ to the cluster.

$\text{AddNewServer}(i, j) \triangleq$

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

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

$\text{DeleteServer}(i, j) \triangleq$

\[
\begin{align*}
\wedge \text{state}[i] &= \text{Leader} \\
\wedge \text{UNCHANGED } \langle \text{messages, serverVars, candidateVars, leaderVars, commitIndex} \rangle
\end{align*}
\]
∧ state[j] ∈ {Follower, Candidate}
∧ j ∈ GetConfig(i)
∧ j ≠ i  TODO: A leader cannot remove itself.
∧ Send([mtype ↦ CheckOldConfig, mterm ↦ currentTerm[i], madd ↦ FALSE, mserver ↦ j, msrource ↦ i, mdest ↦ i])
∧ UNCHANGED (serverVars, candidateVars, leaderVars, logVars)

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

Server i receives a RequestVote request from server j with m.mterm ≤ 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], msrource ↦ 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] =
\[
\begin{align*}
\text{votesResponded}[i] \cup \{ j \} \\
\land \lor \land \text{m.mvoteGranted} \\
\land \text{votesGranted}' = [\text{votesGranted} \ \text{EXCEPT} \ ![i] = \\
\text{votesGranted}[i] \cup \{ j \}] \\
\land \text{voterLog}' = [\text{voterLog} \ \text{EXCEPT} \ ![i] = \\
\text{voterLog}[i] \ @@ (j : m.mlog)] \\
\lor \land \neg \text{m.mvoteGranted} \\
\land \text{UNCHANGED} \ \langle \text{votesGranted}, \ \text{voterLog} \rangle \\
\land \text{Discard}(m) \\
\land \text{UNCHANGED} \ \langle \text{serverVars}, \ \text{votedFor}, \ \text{leaderVars}, \ \text{logVars} \rangle \\
\end{align*}
\]

Server \(i\) receives an \(\text{AppendEntries}\) request from server \(j\) with \(m.mterm \leq \text{currentTerm}[i]\). 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.

\(\text{HandleAppendEntriesRequest}(i, \ j, \ m) \triangleq \)

\[
\text{LET} \ \text{logOk} \triangleq \lor \text{m.mprevLogIndex} = 0 \\
\land \lor \text{m.mprevLogIndex} > 0 \\
\land \lor \text{m.mprevLogTerm} = \text{log}[i][\text{m.mprevLogIndex}].\text{term} \\
\text{IN} \ \land \text{m.mterm} \leq \text{currentTerm}[i] \\
\land \lor \land \text{reject request} \\
\land \lor \text{m.mterm} < \text{currentTerm}[i] \\
\land \lor \text{m.mterm} = \text{currentTerm}[i] \\
\land \text{state}[i] = \text{Follower} \\
\land \neg \text{logOk} \\
\land \text{Reply}(\langle \text{mtype} \mapsto \text{AppendEntriesResponse}, \\
\text{mterm} \mapsto \text{currentTerm}[i], \\
\text{msuccess} \mapsto \text{FALSE}, \\
\text{mmatchIndex} \mapsto 0, \\
\text{msource} \mapsto i, \\
\text{mdest} \mapsto j \rangle, \\
\text{m} ) \\
\land \text{UNCHANGED} \ \langle \text{serverVars}, \ \text{logVars} \rangle \\
\lor \land \text{return to follower state} \\
\land \text{m.mterm} = \text{currentTerm}[i] \\
\land \text{state}[i] = \text{Candidate} \\
\land \text{state}' = [\text{state} \ \text{EXCEPT} \ ![i] = \text{Follower}] \\
\land \text{UNCHANGED} \ \langle \text{currentTerm}, \ \text{votedFor}, \ \text{logVars}, \ \text{messages} \rangle \\
\lor \land \text{accept request} \\
\land \text{m.mterm} = \text{currentTerm}[i] \\
\land \text{state}[i] = \text{Follower} \\
\land \text{logOk} \\
\land \text{LET} \ \text{index} \triangleq \text{m.mprevLogIndex} + 1 \\
\text{IN} \ \lor \land \text{already done with request}
\]
\[ \land \lor m.\text{entries} = \langle \rangle \]
\[ \land \land \log[i]\text{[index].term} = m.\text{entries}[1].\text{term} \]

This could make our commitIndex decrease (for example if we process an old, duplicated request), but that doesn’t really affect anything.

\[ \land \text{commitIndex}' = [\text{commitIndex EXCEPT } ![i] = m.mcommitIndex] \]

\[ \land \text{Reply}([m\text{ntype} \mapsto \rightarrow \text{AppendEntriesResponse}, \]
\[ m\text{term} \mapsto \rightarrow \text{currentTerm}[i], \]
\[ m\text{success} \mapsto \rightarrow \text{TRUE}, \]
\[ m\text{matchIndex} \mapsto \rightarrow m.\text{mprevLogIndex} + \]
\[ \text{Len}(m.\text{entries}), \]
\[ m\text{source} \mapsto \rightarrow i, \]
\[ m\text{dest} \mapsto \rightarrow j], m) \]

\[ \land \text{UNCHANGED } \langle \text{votedFor, currentTerm, log, state} \rangle \]
\[ \lor \land \text{conflict: remove 1 entry} \]
\[ \land \land m.\text{entries} \neq \langle \rangle \]
\[ \land \land \log[i]\text{[index].term} \neq m.\text{entries}[1].\text{term} \]
\[ \land \text{LET new } \leftarrow [\text{index2} \in 1 \ldots (\text{Len}(\log[i]) - 1) \mapsto \log[i]\text{[index2]]}
\]
\[ \land \text{IN } \log' = [\log \text{EXCEPT ![i] = new}] \]
\[ \land \text{UNCHANGED } \langle \text{serverVars, commitIndex, messages} \rangle \]
\[ \lor \land \text{no conflict: append entry} \]
\[ \land \land m.\text{entries} \neq \langle \rangle \]
\[ \land \land \log[i]\text{[index]} = m.\text{mprevLogIndex} \]
\[ \land \land \log' = [\log \text{EXCEPT ![i] = Append(\log[i], m.\text{entries}[1])] \]
\[ \land \text{UNCHANGED } \langle \text{serverVars, commitIndex, messages} \rangle \]
\[ \land \text{UNCHANGED } \langle \text{candidateVars, leaderVars} \rangle \]

Server i receives an AppendEntries response from server j with
\[ m.m\text{term} = \text{currentTerm}[i]. \]

\[ \text{HandleAppendEntriesResponse}(i, j, m) \triangleq \]
\[ \land \land m.m\text{term} = \text{currentTerm}[i] \]
\[ \land \lor \land m.m\text{success} \text{ successful} \]
\[ \land \text{nextIndex}' = [\text{nextIndex EXCEPT ![i][j] = m.m\text{matchIndex} + 1}] \]
\[ \land \text{matchIndex}' = [\text{matchIndex EXCEPT ![i][j] = m.m\text{matchIndex}] \]
\[ \lor \land \neg m.m\text{success} \text{ not successful} \]
\[ \land \text{nextIndex}' = [\text{nextIndex EXCEPT ![i][j] = Max(\{nextIndex[i][j] - 1, 1\})}] \]
\[ \land \text{UNCHANGED } \langle \text{matchIndex} \rangle \]
\[ \land \text{Discard}(m) \]
Detached server $i$ receives a CatchupRequest from leader $j$.

$\text{HandleCatchupRequest}(i, j, m) \triangleq$

$\land \land m.\text{mterm} < \text{currentTerm}[i]$

$\land \land \text{Reply}([\text{mtype} \mapsto \text{CatchupResponse},$

$\text{mterm} \mapsto \text{currentTerm}[i],$

$\text{msuccess} \mapsto \text{FALSE},$

$\text{mmatchIndex} \mapsto 0,$

$\text{msource} \mapsto i,$

$\text{mdest} \mapsto j,$

$\text{mroundsLeft} \mapsto 0],$

$m)$

$\land \land \text{UNCHANGED} \langle \text{serverVars}, \text{candidateVars}, \text{logVars}, \text{elections} \rangle$

$\lor \land m.\text{mterm} \geq \text{currentTerm}[i]$

$\land \land \text{currentTerm}' = \text{currentTerm \ except \ ![i] = m.\text{mterm}}$

$\land \land \text{log}' = \text{log \ except \ ![i] = \text{SubSeq}(\text{log}[i], 1, m.\text{mlogLen}) \circ m.\text{mentries}}$

$\land \land \text{Reply}([\text{mtype} \mapsto \text{CatchupResponse},$

$\text{mterm} \mapsto \text{currentTerm}[i],$

$\text{msuccess} \mapsto \text{TRUE},$

$\text{mmatchIndex} \mapsto \text{Len}(\text{log}[i]),$

$\text{msource} \mapsto i,$

$\text{mdest} \mapsto j,$

$\text{mroundsLeft} \mapsto m.\text{mrounds} - 1],$

$m)$

$\land \land \text{UNCHANGED} \langle \text{serverVars}, \text{candidateVars}, \text{leaderVars}, \text{logVars} \rangle$

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

$\text{HandleCatchupResponse}(i, j, m) \triangleq$

A real system checks for progress every timeout interval.

Assume that if this response is called, the new server has made progress.

$\land \lor \land m.\text{msuccess}$

$\land \lor \land m.\text{mmatchIndex} \neq \text{commitIndex}[i]$  

$\land \lor m.\text{mmatchIndex} \neq \text{matchIndex}[i][j]$  

$\lor m.\text{mmatchIndex} = \text{commitIndex}[i]$

$\land \text{state}[i] = \text{Leader}$

$\land m.\text{mterm} = \text{currentTerm}[i]$

$\land j \notin \text{GetConfig}(i)$

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

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

$\land \lor \land m.\text{mroundsLeft} \neq 0$

$\land \text{Reply}([\text{mtype} \mapsto \text{CatchupRequest},$
mterm \mapsto currentTerm[i],
mentries \mapsto SubSeq(log[i],
nextIndex[i][j],
commitIndex[i]),

mLogLen \mapsto nextIndex[i][j] - 1,
msource \mapsto i,
mdest \mapsto j,
mrounds \mapsto m.mroundsLeft[m],

\bigvee \wedge 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.

\wedge Reply([mtype \mapsto CheckOldConfig,
mterm \mapsto currentTerm[i],
madd \mapsto TRue,
mserver \mapsto j,
msource \mapsto i,
mdest \mapsto i], m)

\wedge UNCHANGED \langle elections \rangle

\bigvee \wedge \neg m.msucceed

\bigvee \bigwedge m.mmatchIndex = commitIndex[i]

\bigvee m.mmatchIndex = matchIndex[i][j]

\wedge m.mmatchIndex \neq commitIndex[i]

\bigvee state[i] \neq Leader

\bigvee m.mterm \neq currentTerm[i]

\bigvee j \in GetConfig(i)

\wedge Discard(m)

\wedge UNCHANGED \langle serverVars, candidateVars, logVars \rangle

Leader i receives a CheckOldConfig message.

\text{HandleCheckOldConfig}(i, m) \triangleq 
\bigvee \wedge state[i] \neq Leader \wedge m.mterm = currentTerm[i]

\wedge Discard(m)

\wedge UNCHANGED \{serverVars, candidateVars, leaderVars, logVars\}

\bigvee \wedge state[i] = Leader \wedge m.mterm = currentTerm[i]

\wedge \wedge \wedge GetMaxConfigIndex(i) \leq commitIndex[i]

\wedge \text{LET newConfig} \triangleq \text{IF} m.madd \text{ THEN UNION } \{GetConfig(i), \{m.mserver\}\}

\text{ELSE GetConfig(i) \backslash \{m.mserver\}\}

newEntry \triangleq [term \mapsto currentTerm[i], type \mapsto ConfigEntry, value \mapsto newConfig]

newLog \triangleq \text{Append}(log[i], newEntry)

\text{IN } log' = [log \text{ EXCEPT } ![i] = newLog]

\wedge Discard(m)
∧ UNCHANGED ⟨commitIndex⟩
∨ ∧ GetMaxConfigIndex(i) > commitIndex[i]
∧ Reply([mtype ↔ CheckOldConfig,  
          mterm ↔ currentTerm[i],
          madd ↔ m.madd,
          mserv ↔ m.mserv,
          msour ↔ i,
          mdest ↔ i],
         m)
∧ UNCHANGED ⟨logVars⟩
∧ UNCHANGED ⟨serverVars, candidateVars, leaderVars⟩

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

\[
\text{UpdateTerm}(i, j, m) \triangleq \\
\land m.mterm > currentTerm[i] \\
\land \text{currentTerm}' = [\text{currentTerm} \text{ EXCEPT } ![i] = m.mterm] \\
\land \text{state}' = [\text{state} \text{ EXCEPT } ![i] = \text{Follower}] \\
\land \text{votedFor}' = [\text{votedFor} \text{ EXCEPT } ![i] = \text{Nil}]
\]

messages is unchanged so \( m \) can be processed further.
∧ UNCHANGED ⟨messages, candidateVars, leaderVars, logVars⟩

Responses with stale terms are ignored.

\[
\text{DropStaleResponse}(i, j, m) \triangleq \\
\land m.mterm < currentTerm[i] \\
\land \text{Discard}(m)
\]
∧ UNCHANGED ⟨serverVars, candidateVars, leaderVars, logVars⟩

Receive a message.

\[
\text{Receive}(m) \triangleq \\
\text{LET } i \triangleq m.mdest \\
\land j \triangleq m.msour
\]

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

\[
\lor \land m.mtype = \text{RequestVoteRequest} \\
\land \text{HandleRequestVoteRequest}(i, j, m)
\lor \land m.mtype = \text{RequestVoteResponse} \\
\land \lor \text{DropStaleResponse}(i, j, m)
\land \text{HandleRequestVoteResponse}(i, j, m)
\lor \land m.mtype = \text{AppendEntriesRequest} \\
\land \text{HandleAppendEntriesRequest}(i, j, m)
\lor \land m.mtype = \text{AppendEntriesResponse} \\
\land \lor \text{DropStaleResponse}(i, j, m)
\land \text{HandleAppendEntriesResponse}(i, j, m)
\lor \land m.mtype = \text{CatchupRequest} \\
\land \text{HandleCatchupRequest}(i, j, m)
\]
∀ m.mtype = CatchupResponse
∧ HandleCatchupResponse(i, j, m)
∀ m.mtype = CheckOldConfig
∧ HandleCheckOldConfig(i, m)

End of message handlers.

Network state transitions

The network duplicates a message
\[ \text{DuplicateMessage}(m) \triangleq \wedge \text{Send}(m) \wedge \text{UNCHANGED } \langle \text{serverVars}, \text{candidateVars}, \text{leaderVars}, \text{logVars} \rangle \]

The network drops a message
\[ \text{DropMessage}(m) \triangleq \wedge \text{Discard}(m) \wedge \text{UNCHANGED } \langle \text{serverVars}, \text{candidateVars}, \text{leaderVars}, \text{logVars} \rangle \]

Model invariants.

Safety property that only a single leader can be elected at a time.
\[ \text{OneLeader} \triangleq \text{Cardinality}(\{ i \in \text{Server} : \text{state}[i] = \text{Leader} \}) \leq 1 \]

Defines how the variables may transition.
\[ \text{Next} \triangleq \wedge \forall \exists i \in \text{Server} : \text{Restart}(i) \]
\[ \forall \exists i \in \text{Server} : \text{Timeout}(i) \]
\[ \forall \exists i, j \in \text{Server} : \text{RequestVote}(i, j) \]
\[ \forall \exists i \in \text{Server} : \text{BecomeLeader}(i) \]
\[ \forall \exists i \in \text{Server}, v \in \text{Value} : \text{ClientRequest}(i, v) \]
\[ \forall \exists i, j \in \text{Server} : \text{AddNewServer}(i, j) \]
\[ \forall \exists i, j \in \text{Server} : \text{DeleteServer}(i, j) \]
\[ \forall \exists i \in \text{Server} : \text{AdvanceCommitIndex}(i) \]
\[ \forall \exists i, j \in \text{Server} : \text{AppendEntries}(i, j) \]
\[ \forall \exists m \in \text{DOMAIN} \text{ messages} : \text{Receive}(m) \]
\[ \forall \exists m \in \text{DOMAIN} \text{ messages} : \text{DuplicateMessage}(m) \]
\[ \forall \exists m \in \text{DOMAIN} \text{ messages} : \text{DropMessage}(m) \]

History variable that tracks every log ever:
\[ \wedge \text{allLogs}' = \text{allLogs} \cup \{ \text{log}[i] : i \in \text{Server} \} \]

The specification must start with the initial state and transition according to Next.
\[ \text{Spec} \triangleq \text{Init} \wedge \Box[\text{Next}]_{\text{vars}} \]
Changelog:

2015 − 05 − 10:
− Add cluster membership changes as described in Section 4 of Diego Ongaro. Consensus: Bridging theory and practice.
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