Safety Proof and Formal Specification for Raft

Diego Ongaro and John Ousterhout
Stanford University

last updated March 28, 2013

1 Introduction

This document is a proof of safety for the Raft replication algorithm. Liveness is not addressed, nor is reconfiguration.

2 System model

This document uses the usual asynchronous system assumptions.

- Messages may take an arbitrary number of steps to arrive at a server; once they arrive, they are modeled as being processed in one atomic step.
- Servers fail by stopping and may later restart from stable storage on disk.
- The network may reorder, drop, and duplicate messages.

3 Conventions

The specification uses the syntax and semantics of the TLA+ language version 2. The remainder of this document uses the same syntax and semantics but with the following minor allowances for convenience.

- As in TLA+, foo' has a specific meaning: the value of variable *foo* in the next state of the system.
- We say $\langle index, term \rangle \in log$ iff
  $Len(log) \geq index \;\wedge\; log[index].term = term$.
- We use the symbol $\|$ for concatenation of logs and entries.
- We ignore values in log entries, since it's easy to see that a value is attached to a particular $\langle index, term \rangle$, and those uniquely identify a log entry.

4 Specification

This section provides a complete, formal description of the Raft algorithm.

MODULE *raft*

This is the formal specification for the Raft replication algorithm.

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

The set of server *IDs*

```
CONSTANTS Replica

  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

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

----------------------------------------------------------------------------
  The following variables are all per server (functions with the domain Replica).

  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
replicaVars ≜ ⟨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 latest entry the state machine may apply is commitIndex.
VARIABLE commitIndex
logVars ≜ ⟨log, commitIndex⟩

  The following variables are used only on candidates:
  The set of servers from which the candidate has received a RequestVote response in this term.
VARIABLE votesResponded
  The set of servers from which the candidate has received a vote in this term.
VARIABLE votesGranted
  A history variable used in the proof. This would not be present in an implementation.
  Keeps track of the log of each voter.
VARIABLE voterLog
candidateVars ≜ ⟨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 $lastAgreeIndex$

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

End of per server variables.

All variables; used for stuttering.

$vars \triangleq \langle messages,\ replicaVars,\ candidateVars,\ leaderVars,\ logVars \rangle$

Helpers

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

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

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*

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

Helper for *Discard* and *Reply*

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

Add a message to the set of messages.

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

The recipient is done processing the message.

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

Combination of *Send* and *Discard*

$Reply(response,\ request) \triangleq$
$\quad messages' = WithoutMessage(request,\ WithMessage(response,\ messages))$

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.

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

Define initial values for all variables

```
InitHistoryVars ≜ ∧ elections = {}
                  ∧ allLogs   = {}
                  ∧ voterLog = [i ∈ Replica ↦ [j ∈ {} ↦ ⟨⟩]]
InitReplicaVars ≜ ∧ currentTerm = [i ∈ Replica ↦ 1]
                  ∧ state       = [i ∈ Replica ↦ Follower]
                  ∧ votedFor    = [i ∈ Replica ↦ Nil]
InitCandidateVars ≜ ∧ votesResponded = [i ∈ Replica ↦ {}]
                    ∧ votesGranted   = [i ∈ Replica ↦ {}]
 The values nextIndex[i][i] and lastAgreeIndex[i][i] are never read, since
 the leader does not send itself messages. It's still easier to include these
 in the functions.
InitLeaderVars ≜ ∧ nextIndex      = [i ∈ Replica ↦ [j ∈ Replica ↦ 1]]
                 ∧ lastAgreeIndex = [i ∈ Replica ↦ [j ∈ Replica ↦ 0]]
InitLogVars ≜ ∧ log         = [i ∈ Replica ↦ ⟨⟩]
              ∧ commitIndex = [i ∈ Replica ↦ 0]
Init ≜ ∧ messages = [m ∈ {} ↦ 0]
       ∧ InitHistoryVars
       ∧ InitReplicaVars
       ∧ InitCandidateVars
       ∧ InitLeaderVars
       ∧ InitLogVars

 The network duplicates a message
DuplicateMessage ≜ ∧ ∃ m ∈ DOMAIN messages :
                        Send(m)
                   ∧ UNCHANGED ⟨replicaVars, candidateVars, leaderVars, logVars⟩

 The network drops a message
DropMessage ≜ ∧ ∃ m ∈ DOMAIN (messages) :
                   Discard(m)
              ∧ UNCHANGED ⟨replicaVars, candidateVars, leaderVars, logVars⟩

 Server i times out and starts a new election.
Timeout(i) ≜ ∧ state[i] ∈ {Follower, Candidate}
             ∧ state' = [state EXCEPT ![i] = Candidate]
             ∧ currentTerm' = [currentTerm EXCEPT ![i] = currentTerm[i] + 1]
              Most implementations would probably just set the local vote
              atomically, but messaging localhost for it is weaker.
             ∧ votedFor' = [votedFor EXCEPT ![i] = Nil]
             ∧ votesResponded' = [votesResponded EXCEPT ![i] = {}]
             ∧ votesGranted'   = [votesGranted EXCEPT ![i] = {}]
             ∧ voterLog'       = [voterLog EXCEPT ![i] = [j ∈ {} ↦ ⟨⟩]]
             ∧ UNCHANGED ⟨messages, leaderVars, logVars⟩

 Server i restarts from stable storage.
 It loses everything but its currentTerm, votedFor, and log.
Restart(i) ≜ ∧ UNCHANGED ⟨messages, currentTerm, votedFor, log, elections⟩
             ∧ state'          = [state EXCEPT ![i] = Follower]
             ∧ votesResponded' = [votesResponded EXCEPT ![i] = {}]
             ∧ votesGranted'   = [votesGranted EXCEPT ![i] = {}]
             ∧ voterLog'       = [voterLog EXCEPT ![i] = [j ∈ {} ↦ ⟨⟩]]
             ∧ nextIndex'      = [nextIndex EXCEPT ![i] = [j ∈ Replica ↦ 1]]
             ∧ lastAgreeIndex' = [lastAgreeIndex EXCEPT ![i] = [j ∈ Replica ↦ 0]]
```

```
                  ∧ commitIndex'    = [commitIndex EXCEPT ![i] = 0]

 Candidate i sends j a RequestVote request.
RequestVote(i, j) ≜ ∧ state[i] = Candidate
                    ∧ j ∉ votesResponded[i]
                    ∧ Send([mtype         ↦ RequestVoteRequest,
                            mterm         ↦ currentTerm[i],
                            mlastLogTerm  ↦ LastTerm(log[i]),
                            mlastLogIndex ↦ Len(log[i]),
                            msource       ↦ i,
                            mdest         ↦ j])
                    ∧ UNCHANGED ⟨replicaVars, candidateVars, leaderVars, logVars⟩

 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) ≜
    ∧ i ≠ j
    ∧ state[i] = Leader
    ∧ LET prevLogIndex ≜ nextIndex[i][j] − 1
          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] + 1})
          entries ≜ SubSeq(log[i], nextIndex[i][j], upper)
      IN  Send([mtype          ↦ AppendEntriesRequest,
                mterm          ↦ currentTerm[i],
                mprevLogIndex  ↦ prevLogIndex,
                mprevLogTerm   ↦ prevLogTerm,
                mentries       ↦ entries,
                mcommitIndex   ↦ Min({commitIndex[i], lastEntry}),
                msource        ↦ i,
                mdest          ↦ j])
    ∧ UNCHANGED ⟨replicaVars, candidateVars, leaderVars, logVars⟩

 Candidate i transitions to leader.
BecomeLeader(i) ≜ ∧ state[i] = Candidate
                  ∧ votesGranted[i] ∈ Quorum
                  ∧ state' = [state EXCEPT ![i] = Leader]
                  ∧ nextIndex'      = [nextIndex EXCEPT ![i] =
                                         [j ∈ Replica ↦ Len(log[i]) + 1]]
                  ∧ lastAgreeIndex' = [lastAgreeIndex EXCEPT ![i] =
                                         [j ∈ Replica ↦ 0]]
                  ∧ elections' = elections ∪
                                   {[e.eterm      ↦ currentTerm[i],
                                     e.eleader    ↦ i,
                                     e.elog       ↦ log[i],
                                     e.evotes     ↦ votesGranted[i],
                                     e.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]
                            newIndex ≜ Len(log[i]) + 1
                             newLog ≜ Append(log[i], entry)
                        IN   log′ = [log EXCEPT ![i] = newLog]
                      ∧ UNCHANGED ⟨messages, replicaVars, candidateVars, leaderVars,
                                   commitIndex⟩

────────────────────────────────────────────────────────────────────────────────
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 logIsCurrent ≜ ∨ m.mlastLogTerm > LastTerm(log[i])
                       ∨ ∧ m.mlastLogTerm = LastTerm(log[i])
                         ∧ m.mlastLogIndex ≥ Len(log[i])
        grant ≜ ∧ m.mterm = currentTerm[i]
                ∧ logIsCurrent
                ∧ 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) ≜
    ∧ m.mterm = currentTerm[i]
    ∧ votesResponded′ = [votesResponded EXCEPT ![i] = votesResponded[i] ∪ {j}]
    ∧ ∨ m.mvoteGranted  ∧ votesGranted′ = [votesGranted EXCEPT ![i] = votesGranted[i] ∪ {j}]
                        ∧ voterLog′ = [voterLog EXCEPT ![i][j] = m.mlog]
      ∨ ¬m.mvoteGranted ∧ UNCHANGED ⟨votesGranted, voterLog⟩
    ∧ Discard(m)
    ∧ UNCHANGED ⟨replicaVars, votedFor, leaderVars, logVars⟩

Server i receives an AppendEntries request from server j with m.mterm ≤ 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 accept ≜ ∧ m.mterm = currentTerm[i]
                 ∧ ∨ 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
              ∧ ¬accept
              ∧ Reply([mtype            ↦ AppendEntriesResponse,
                       mterm            ↦ currentTerm[i],
                       mlastAgreeIndex  ↦ 0,
                       msource          ↦ i,
                       mdest            ↦ j],
                       m)
              ∧ UNCHANGED ⟨replicaVars, candidateVars, leaderVars, logVars⟩
           ∨ ∧ accept
              ∧ 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],
                                  mlastAgreeIndex  ↦ m.mprevLogIndex + Len(m.mentries),
                                  msource          ↦ i,
                                  mdest            ↦ j],
                                  m)
                         ∧ UNCHANGED ⟨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 ⟨commitIndex, messages⟩
                      ∨ no conflict: add entry
                         ∧ m.mentries ≠ ⟨⟩
                         ∧ Len(log[i]) = m.mprevLogIndex
                         ∧ log' = [log EXCEPT ![i] = Append(log[i], m.mentries[1])]
                         ∧ UNCHANGED ⟨commitIndex, messages⟩
         ∧ UNCHANGED ⟨replicaVars, 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.mlastAgreeIndex > 0
         ∧ nextIndex'       = [nextIndex       EXCEPT ![i][j] = m.mlastAgreeIndex + 1]
         ∧ lastAgreeIndex'  = [lastAgreeIndex  EXCEPT ![i][j] = m.mlastAgreeIndex]
         ∧ LET  Agree(index) is the set of replicas that agree up through index.
                Agree(index) ≜ {i} ∪ {k ∈ Replica : lastAgreeIndex'[i][k] ≥ index}
                 The set of 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]
        ∨ ∧ m.mlastAgreeIndex = 0
          ∧ nextIndex' = [nextIndex EXCEPT ![i][j] = Max({nextIndex[i][j] − 1, 1})]
          ∧ UNCHANGED ⟨lastAgreeIndex, commitIndex⟩
    ∧ Discard(m)
    ∧ UNCHANGED ⟨replicaVars, candidateVars, log, elections⟩

 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 ⟨replicaVars, candidateVars, leaderVars, logVars⟩

 Receive a message.
Receive ≜ ∃ m ∈ DOMAIN messages :
            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)

 End of message handlers.
────────────────────────────────────────────────────────────────────────────

 Defines how the variables may transition.
Next ≜ ∧ ∨ DuplicateMessage
         ∨ DropMessage
         ∨ Receive
         ∨ ∃ i ∈ Replica : Timeout(i)
         ∨ ∃ i ∈ Replica : Restart(i)
         ∨ ∃ i ∈ Replica : BecomeLeader(i)
         ∨ ∃ i ∈ Replica, v ∈ Value : ClientRequest(i, v)
```

$$
\begin{aligned}
&\qquad\qquad \lor \exists\, i, j \in \mathit{Replica} : \mathit{RequestVote}(i, j) \\
&\qquad\qquad \lor \exists\, i, j \in \mathit{Replica} : \mathit{AppendEntries}(i, j)
\end{aligned}
$$

History variable that tracks every *log* ever:

$$\qquad \land\ \mathit{allLogs}' = \mathit{allLogs} \cup \{\forall\, i \in \mathit{Replica} : \mathit{log}[i]\}$$

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

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

5 Definitions

Definition 1. An entry $\langle \mathit{index}, \mathit{term} \rangle$ is **committed at term** t if it is present in every leader's log following t:

$$
\begin{aligned}
\mathit{committed}(t) \triangleq \{\langle \mathit{index}, \mathit{term} \rangle :\ & \\
&\forall\ \mathit{election} \in \mathit{elections} : \\
&\quad \mathit{election.term} > t \Rightarrow \\
&\qquad \langle \mathit{index}, \mathit{term} \rangle \in \mathit{election.log}\}
\end{aligned}
$$

Definition 2. An entry $\langle \mathit{index}, \mathit{term} \rangle$ is **immediately committed** if it is acknowledged by a quorum (including the leader) during *term*. Theorem 9 shows that these entries are *committed at term*.

$$
\begin{aligned}
\mathit{immediatelyCommitted} \triangleq \{\langle \mathit{index}, \mathit{term} \rangle :\ & \\
&\exists\ \mathit{leader}, \mathit{subquorum} : \\
&\quad \land\ \mathit{subquorum} \cup \{\mathit{leader}\} \in \mathit{Quorum} \\
&\quad \land\ \forall\ i \in \mathit{subquorum} : \\
&\qquad\quad \exists\ m \in \mathit{messages} : \\
&\qquad\qquad \land\ m.\mathit{msource} = i \\
&\qquad\qquad \land\ m.\mathit{mdest} = \mathit{leader} \\
&\qquad\qquad \land\ m.\mathit{mtype} = \mathit{AppendEntriesResponse} \\
&\qquad\qquad \land\ m.\mathit{mterm} = \mathit{term} \\
&\qquad\qquad \land\ m.\mathit{mlastAgreeIndex} \geq \mathit{index}\}
\end{aligned}
$$

Definition 3. An entry $\langle \mathit{index}, \mathit{term} \rangle$ is **prefix committed at term** t if there is another entry that is *committed at term* t following it in some log. Theorem 10 shows that these entries are *committed at term* t.

$$
\begin{aligned}
\mathit{prefixCommitted}(t) \triangleq \{\langle \mathit{index}, \mathit{term} \rangle :\ & \\
&\exists\ i \in \mathit{Server} : \\
&\quad \land\ \langle \mathit{index}, \mathit{term} \rangle \in \mathit{log}[i] \\
&\quad \land\ \exists\ \langle \mathit{rindex}, \mathit{rterm} \rangle \in \mathit{log}[i] : \\
&\qquad\quad \land\ \mathit{index} < \mathit{rindex} \\
&\qquad\quad \land\ \langle \mathit{rindex}, \mathit{rterm} \rangle \in \mathit{committed}(t)\}
\end{aligned}
$$

6 Proof

Lemma 1. Each server's *currentTerm* monotonically increases:

$$
\begin{aligned}
&\forall\ i \in \mathit{Server} : \\
&\quad \mathit{currentTerm}[i] \leq \mathit{currentTerm}'[i]
\end{aligned}
$$

Proof. This follows immediately from the specification. □

Lemma 2. There is at most one leader per term:

$$\begin{aligned}&\forall\ e, f \in \mathit{elections} :\\ &\quad e.\mathit{term} = f.\mathit{term} \Rightarrow e = f\end{aligned}$$

Sketch. It takes votes from a quorum to become leader, voters may only vote once per term, and any two quorums overlap. Moreover, no single server starts multiple elections in the same term.

Proof.

1. Consider two elections, e and f, both members of *elections*, where $e.\mathit{term} = f.\mathit{term}$.
2. $e.\mathit{votes} \in \mathit{Quroum}$ and $f.\mathit{votes} \in \mathit{Quorum}$, since this is a necessary condition for elements of *elections*.
3. Let *voter* be an arbitrary member of $e.\mathit{votes} \cap f.\mathit{votes}$. Such a member must exist since any two quorums overlap.
4. Once *voter* casts a vote for $e.\mathit{leader}$ in $e.\mathit{term}$, it can not cast a vote for a different server in $e.\mathit{term}$ (the specification ensures this: once it increments its *currentTerm*, it can never vote again for the same server (Lemma 1); and until then, it safely retains its vote information).
5. $e.\mathit{leader} = f.\mathit{leader}$, since *voter* voted for $e.\mathit{leader}$ and *voter* voted for $f.\mathit{leader}$ in $e.\mathit{term} = f.\mathit{term}$.
6. A server does not start more than one election in the same term, since it increments its term upon starting a new election, and its *currentTerm* monotonically increases.
7. Therefore, $e = f$.

□

Lemma 3. A leader's log monotonically grows during its term:

$$\begin{aligned}&\forall\ e \in \mathit{elections} :\\ &\qquad \mathit{currentTerm}[e.\mathit{leader}] = e.\mathit{term} \Rightarrow\\ &\qquad\qquad \forall\ \mathit{index} \in \mathit{Len}(\mathit{log}[e.\mathit{leader}]) :\\ &\qquad\qquad\qquad \mathit{log}'[e.\mathit{leader}][\mathit{index}] = \mathit{log}[e.\mathit{leader}][\mathit{index}]\end{aligned}$$

Sketch. While $\mathit{state}[i] = \mathit{Leader}$, i only appends to its log. i won't ever get an AppendEntries request from some other server for this term, since there is at most one leader per term. And i rejects AppendEntries requests for other terms until increasing term.

Proof.

1. Three variables are involved: *elections*, *currentTerm*, and *log*.
2. When a new election is added to *elections*, the *log* of the leader is not changed in the same step, so the invariant is maintained.
3. Logs can change either from client requests or AppendEntries requests:
   (a) Case: client request: follows directly from the spec
   (b) Case: AppendEntries request
      i. Only servers with $\mathit{state}[i] = \mathit{Leader}$ can send AppendEntries requests for their *currentTerm*.
      ii. By Lemma 2, $e.\mathit{leader}$ is the only leader for $e.\mathit{term}$.
      iii. Servers don't send themselves AppendEntries requests (see spec).
      iv. So $e.\mathit{leader}$ will receive no AppendEntries requests during $e.\mathit{term}$.

4. $currentTerm[e.leader]$ monotonically increases by Lemma 1, so once $e.leader$ moves to a new term, it will trivially satisfy the invariant forever after.

$\square$

Lemma 4. An $\langle index, term \rangle$ identifies a log prefix:

$$
\begin{aligned}
&\forall\ l, m \in allLogs : \\
&\quad \forall\ index, term : \\
&\qquad \langle index, term \rangle \in l \land \langle index, term \rangle \in m \Rightarrow \\
&\qquad\quad \forall\ pindex \in 1..index : \\
&\qquad\qquad\quad l[pindex] = m[pindex]
\end{aligned}
$$

Sketch. Only leaders create entries, and they assign the new entries term numbers that will never be assigned again by other leaders (there's at most one leader per term). The consistency check in AppendEntries guarantees that when followers accept new entries, they do so in a way that's consistent with the leader's log at the time it sent the entries.

Assertion. If $log'[i]$ is a prefix of some log in $allLogs$, then $allLogs' = allLogs \cup \{log'[i]\}$ maintains the invariant.

Proof by induction on an execution.

1. Initial state: all logs are empty, so the invariant holds.

2. Inductive step: logs change by either:

   (a) Case: a leader adds one entry (client request)

      i. This $\langle index, term \rangle$ uniquely identifies this entry, since there's only one leader per term (Lemma 2) and leaders only append to their logs.

      ii. By the inductive hypothesis, $log[leader] \in allLogs$.

      iii. Then $allLogs' = allLogs \cup \{log[leader] \parallel \langle index, term \rangle\}$ maintains the invariant.

   (b) Case: a follower, $follower$, removes one entry (AppendEntries request m)

      i. The invariant still holds, since $log'[follower]$ is a prefix of $log[follower]$ (by the Assertion above).

   (c) Case: a follower, $follower$, adds one entry (AppendEntries request m)

      i. Let $l \triangleq log[m.msource]$ at the time the leader creates the AppendEntries request.

      ii. $l \in allLogs$ by definition of $allLogs$.

      iii. In the two cases below, we show that $log'[follower]$ is a prefix of l. By the Assertion above, this suffices to show that the invariant is maintained.

      iv. Case: $m.mentries$ is a prefix of l.

         A. $m.mprevLogIndex = 0$, since nothing precedes $m.mentries$ in l.

         B. $log[follower]$ is empty, as a necessary condition for accepting the request.

         C. $log'[follower] = m.mentries$ upon accepting the request, which is a prefix of l.

      v. Case: $start \parallel \langle m.mprevLogIndex, m.mprevLogTerm \rangle \parallel m.mentries$ is a prefix of l, where $start$ is some (possibly empty) log prefix.

         A. The follower accepts the request by assumption, so it contains the entry $\langle m.mprevLogIndex, m.mprevLogTerm \rangle$.

         B. By the inductive hypothesis, $log[follower]$ contains the prefix $start \parallel \langle m.mprevLogIndex, m.mprevLogTerm \rangle$.

         C. $follower'[log] = start \parallel \langle m.mprevLogIndex, m.mprevLogTerm \rangle \parallel m.mentries$ upon accepting the request, which is a prefix of l.

$\square$

Lemma 5. A server's current term is always at least as large as the terms in its log:

$$\begin{aligned}&\forall\, i \in \mathit{Server} :\\ &\quad \forall\, \mathit{index} \in 1..\mathit{Len}(\mathit{log}[i]) :\\ &\quad\quad \mathit{log}[i][\mathit{index}].\mathit{term} \leq \mathit{currentTerm}[i]\end{aligned}$$

Sketch. Servers' current terms monotonically increase. When leaders create new entries, they assign them their current term. And when followers accept new entries from a leader, they agree with the leader's term at the time it sent the entries.

Proof by induction on an execution.

1. Initial state: all logs are empty, so the invariant holds.
2. Inductive step: logs change by either:
   (a) Case: a leader adds one entry (client request)
      i. By the inductive hypothesis, all prior entries in $\mathit{log}[\mathit{leader}]$ have term $\leq \mathit{currentTerm}[\mathit{leader}]$.
      ii. The new entry's term is $\mathit{currentTerm}[\mathit{leader}]$.
   (b) Case: a follower removes one entry (AppendEntries request)
      i. The invariant still holds, since only the length of the log decreased.
   (c) Case: a follower adds one entry (AppendEntries request m)
      i. By the inductive hypothesis, $\mathit{currentTerm}[\mathit{leader}]$ was at least as large as the term in every entry of $\mathit{log}[\mathit{leader}]$ when leader created the request.
      ii. The leader's log contained $m.\mathit{mentries}$ and $m.\mathit{mterm} = \mathit{currentTerm}[\mathit{leader}]$ at the time leader created the request.
      iii. $\mathit{follower}'[\mathit{log}]$ ends in $m.\mathit{mentries}$ upon accepting the request.
      iv. The leader's log contained every entry present in $\mathit{follower}'[\mathit{log}]$ when it created the request, since they both have $m.\mathit{mentries}$ in common (by Lemma 4).
      v. As a necessary condition for accepting the request, $\mathit{currentTerm}[\mathit{follower}] = m.\mathit{mterm}$.
      vi. Then $\mathit{currentTerm}[\mathit{follower}]$ is at least as large as the term in every entry in $\mathit{log}'[\mathit{follower}]$, and the invariant is maintained.
3. Inductive step: $\mathit{currentTerm}[i]$ changes
   (a) By Lemma 1, $\mathit{currentTerm}'[i] \geq \mathit{currentTerm}[i]$, so the invariant is maintained.

$\square$

Lemma 6. The terms of indexes grow monotonically in a server's log:

$$\begin{aligned}&\forall\, i \in \mathit{Server} :\\ &\quad \forall\, \mathit{index} \in 1..(\mathit{Len}(\mathit{log}[i]) - 1) :\\ &\quad\quad \mathit{log}[i][\mathit{index}].\mathit{term} \leq \mathit{log}[i][\mathit{index}+1].\mathit{term}\end{aligned}$$

Sketch. A leader maintains this by assigning new entries its current term, which is always at least as large as the terms in its log. When followers accept new entries, they are consistent with the leader's log at the time it sent the entries.

Proof by induction on an execution.

1. Initial state: all logs are empty, so the invariant holds.

2. Inductive step: logs change by either:

    (a) Case: a leader adds one entry (client request)

        i. The new entry's term is $currentTerm[leader]$

        ii. $currentTerm[leader]$ is at least as large as the term of any entry in $log[leader]$, by Lemma 5.

    (b) Case: a follower removes one entry (AppendEntries request)

        i. The invariant still holds, since only the length of the log decreased.

    (c) Case: a follower adds one entry (AppendEntries request)

        i. $follower'[log]$ ends in $m.mentries$ upon accepting the request.

        ii. Then $follower'[log]$ is a prefix of the leader's log at the time the leader created the request, since $m.mentries$ identifies a log prefix (Lemma 4).

        iii. By the inductive hypothesis, since the leader's log contained $m.mentries$ at the time it created the request, those entries and all preceding entries satisfy the invariant (have monotonically increasing terms).

□

Lemma 7. If two logs have the same last term, at least one is a prefix of the other:

$$\begin{aligned}
&\forall\, l, m \in \mathit{allLogs} : \\
&\qquad \mathit{LastTerm}(m) = \mathit{LastTerm}(l) \wedge \mathit{Len}(l) \leq \mathit{Len}(m) \Rightarrow \\
&\qquad\quad \forall\, \mathit{index} \in 1..\mathit{Len}(l) : \\
&\qquad\qquad l[\mathit{index}].\mathit{term} = m[\mathit{index}].\mathit{term}
\end{aligned}$$

Sketch. When a leader adds its first entry in a term, its term is distinct from all other entries' terms (there is at most one leader per term), so this trivially holds. When it adds subsequent entries under that term, it only appends those, so its log satisfies this property. When followers accept new entries, they are consistent with the leader's log at the time it sent the entries.

Proof by induction on an execution.

1. Initial state: trivially holds for empty logs.

2. Inductive step: logs change by either:

    (a) Case: a leader adds one entry (client request)

        i. This $\langle index, term\rangle$ uniquely identifies a log entry not previously found in any of $allLogs$, since there is at most one leader per term (Lemma 2), leaders only append entries with terms set to $currentTerm[i]$, and $currentTerm[i]$ monotonically increases (Lemma 1).

        ii. Case: $term = LastTerm(log[leader])$: The invariant holds since $log[leader]$ is a prefix of $log'[leader]$.

        iii. Case: $LastTerm(log'[leader]) > LastTerm(log[leader])$ The invariant holds, since $\forall\, n \in allLogs : LastTerm(log'[leader]) \neq LastTerm(n)$ (there is at most one leader per term by Lemma 2).

    (b) Case: a follower removes one entry (AppendEntries request)

        i. The invariant still holds, since $log'[follower]$ is a prefix of $log[follower]$.

    (c) Case: a follower adds one entry (AppendEntries request m)

        i. The leader's log contained $m.mentries$ at the time it creates the request.

        ii. $log'[follower]$ ends in $m.mentries$ upon applying the request.

        iii. By Lemma 4, $log'[follower]$ is a prefix of the leader's log at the time it created the request, which is in $allLogs$ by the inductive hypothesis.

□

Lemma 8. A leader does not send *AppendEntries* requests that conflict with its log.

$$
\begin{array}{l}
\forall\ i \in \mathit{Server} : \\
\quad (\land\ \mathit{state}[i] = \mathit{Leader} \\
\quad\ \ \land\ \langle \mathit{index}, \mathit{term} \rangle \in \mathit{log}[i] \\
\quad) \Rightarrow \neg \exists\ m \in \text{DOMAIN } \mathit{messages} : \\
\qquad\qquad \land\ m.\mathit{mtype} = \mathit{AppendEntriesRequest} \\
\qquad\qquad \land\ m.\mathit{mterm} = \mathit{currentTerm}[i] \\
\qquad\qquad \land\ \lor\ \langle \mathit{index}, \mathit{otherTerm} \rangle \in m.\mathit{mentries} \\
\qquad\qquad\quad\ \lor\ m.\mathit{mprevLogIndex} = \mathit{index}\ \land\ m.\mathit{mprevLogTerm} = \mathit{otherTerm} \\
\qquad\qquad \land\ \mathit{otherTerm} \neq \mathit{term}
\end{array}
$$

Sketch. Leaders only send AppendEntries using the contents of their log, and a leader's log monotonically grows.

Proof.

1. Since there is only one leader per term, the request would have to come from this server.

2. Leaders only send *AppendEntries* requests that are consistent with their logs.

3. By Lemma 3, a leader's log monotonically grows during its term.

4. Therefore, no server could have sent such a conflicting request.

□

Lemma 9. *Immediately committed* entries are *committed*:

$$
\begin{array}{l}
\forall\ \langle \mathit{index}, \mathit{term} \rangle \in \mathit{immediatelyCommitted} : \\
\quad \langle \mathit{index}, \mathit{term} \rangle \in \mathit{committed}(\mathit{term})
\end{array}
$$

Sketch. A quorum contains the entry in the term when it is created, so all leaders in future terms must receive a vote from at least one of these servers. By the log consistency check during leader election, the first leader following term must have the entry: it couldn't have a larger last log term than the voter's, so the voter's log must be a prefix of the leader's log. The next leader after that could have a larger term, but only if it inherited its log from this leader, so the initial log for this leader (which contains the entry) is a prefix of this next leader's log. This argument continues for all future leaders.

Proof.

1. Consider an entry $\langle \mathit{index}, \mathit{term} \rangle$ that is *immediately committed*.

2. Define

$$
\begin{array}{l}
\mathit{Contradicting} \triangleq \{\mathit{election} \in \mathit{elections} : \\
\qquad\qquad\qquad\qquad \land\ \mathit{election}.\mathit{term} > \mathit{term} \\
\qquad\qquad\qquad\qquad \land\ \langle \mathit{index}, \mathit{term} \rangle \notin \mathit{election}.\mathit{log}\}
\end{array}
$$

3. Let *election* be an element in *Contradicting* with a minimal *term* field. That is, $\forall\ e \in \mathit{Contradicting} : \mathit{election}.\mathit{term} \leq e.\mathit{term}$.

4. It suffices to show a contradiction, which implies $\mathit{Contradicting} = \phi$.

5. Let *follower* be any server that both votes in *election* and contains $\langle \mathit{index}, \mathit{term} \rangle$ during *term* (either it acknowledges the entry as a follower or it was leader). Such a server must exist since:

(a) A quorum votes in *election*.

(b) A quorum contains $\langle index, term \rangle$ during *term*.

(c) Any two quorums overlap.

6. Let $followerLog \triangleq election.voterLog[follower]$.

7. The follower contains the entry when it cast its vote during $election.term$. That is, $\langle index, term \rangle \in followerLog$:

   (a) $\langle index, term \rangle$ was in the follower's log during *term*.

   (b) The follower must have stored the entry in *term* before voting in $election.term$, since $election.term > term$, $currentTerm[follower]$ monotonically increases (Lemma 1), and the follower rejects requests with terms smaller than $currentTerm[follower]$.

   (c) The follower couldn't have removed the entry before casting its vote:

      i. Case: No *AppendEntriesRequest* with $mterm < term$ removes the entry from the follower's log, since $currentTerm[follower] \geq term$ upon storing the entry (by Lemma 5), and the follower rejects requests with terms smaller than $currentTerm[follower]$.

      ii. Case: No *AppendEntriesRequest* with $mterm = term$ removes the entry from the follower's log, since:

         A. There is only one leader of *term*.

         B. The leader of *term* created and therefore contains the entry (Lemma 3).

         C. The leader would not send any conflicting requests to *follower* during *term* (Lemma 8).

      iii. Case: No *AppendEntriesRequest* with $election.term > mterm > term$ removes the entry from the follower's log. Since all of these leaders also contained the entry by assumption, they did not send any conflicting entries to the follower for this index (Lemma 8). Nor did they send any conflicting entries for prior indexes: that they have this entry implies they have the entire prefix (Lemma 4).

      iv. Case: No *AppendEntriesRequest* with $mterm = election.term$ removes the entry from the follower's log prior to it voting, since there is at most one leader per term (Lemma 2), so this request would have to come from the leader of $election.term$, but it hasn't been elected yet.

      v. Case: No *AppendEntriesRequest* with $mterm > election.term$ removes the entry from the follower's log prior to it voting, since then $currentTerm[follower] > election.term$ would prevent the follower from voting in *term*.

8. The log completeness check during elections states the following, since *follower* granted its vote during *election*:

$$
\begin{array}{l}
\lor\ LastTerm(election.log) > LastTerm(followerLog) \\
\lor\ \ \land\ LastTerm(election.log) = LastTerm(followerLog) \\
\quad\ \ \land\ Len(election.log) \geq Len(followerLog)
\end{array}
$$

   In the following two steps, we take each of these cases in turn and show a contradiction.

9. Case: $LastTerm(election.log) > LastTerm(followerLog)$

   (a) $LastTerm(followerLog) \geq term$, since $\langle index, term \rangle \in followerLog$ and terms in logs grow monotonically (Lemma 6).

   (b) $election.term > LastTerm(election.log)$ since servers increment their *currentTerm* when starting an election, and Lemma 5 states that a server's *currentTerm* is at least as large as the terms in its log.

(c) Let *prior* be the election in *elections* with $prior.term = LastTerm(election.log)$. Such an election must exist since $LastTerm(election.log) > 0$ and a server must win an election before creating an entry.

(d) By transitivity, we now have the following inequalities:

$$\begin{aligned}&term \leq \\ &\quad LastTerm(followerLog) < \\ &\quad\quad LastTerm(election.log) = prior.term < \\ &\quad\quad\quad election.term\end{aligned}$$

(e) $\langle index, term\rangle \notin prior.log$, since:

   i. $prior.log$ is a prefix of $election.log$ by Lemma 7, since:

      A. $LastTerm(election.log) = prior.term$

      B. The leader of $prior.term$ creates entries by appending them to $prior.log$, so any entry with term $prior.term$ must have index greater than $Len(prior.log)$.

      C. Then $Len(election.log) > Len(prior.log)$.

   ii. Since $\langle index, term\rangle \notin election.log$ (by assumption), either $Len(election.log) < index$ or $election.log[index].term$ $term$. We consider these cases separately next.

   iii. Case: $Len(election.log) < index$

      A. Then $\langle index, term\rangle \notin prior.log$, since $prior.log$ is a prefix of $election.log$.

   iv. Case: $election.log[index].term \neq term$

      A. Case: $Len(prior.log) < index$: immediate.

      B. Case: $Len(prior.log) > index$: $election.log[index].term = prior.log[index].term \neq term$, since $prior.log$ is a prefix is of $election.log$.

(f) $prior \in Contradicting$, by definition of *Contradicting*.

(g) *election* was defined as an element in *Contradicting* with a minimal *term* field, but *prior* is also in *Contradicting* and $prior.term < election.term$.

10. Case: $LastTerm(election.log) = LastTerm(followerLog)$ and $Len(election.log) \geq Len(followerLog)$

   (a) *followerLog* is a prefix of *election.log* by Lemma 7.

   (b) Then $\langle index, term\rangle \in election.log$, since $\langle index, term\rangle \in followerLog$.

   (c) But $election \in Contradicting$ implies that $\langle index, term\rangle \notin election.log$.

□

Lemma 10. *Prefix committed* entries are *committed*:

$\forall\ t : prefixCommitted(t) \subseteq committed(t)$

Sketch. If an entry is committed, it identifies a prefix of a log in which every entry is committed, since those entries will also be present in every future leader's log.

Proof.

1. Consider an arbitrary entry $\langle index, term\rangle \in prefixCommitted(t)$.

2. There exists an entry $\langle rindex, rterm\rangle \in committed(t)$ following $\langle index, term\rangle$ in some log, by definition of $prefixCommitted(t)$.

3. $\langle rindex, rterm\rangle$ uniquely identifies the log prefix containing $\langle index, term\rangle$ (Lemma 4).

4. Every leader following t contains $\langle \mathit{index}, \mathit{term} \rangle$, since every leader following t contains $\langle \mathit{rindex}, \mathit{rterm} \rangle$.

5. $\langle \mathit{index}, \mathit{term} \rangle \in \mathit{committed}(t)$ by definition of $\mathit{committed}(t)$.

□

Theorem 1. Servers only apply entries that are *committed* in their current term:

$$\begin{array}{l} \forall\ i \in \mathit{Server} : \\ \quad \{\langle \mathit{index}, \mathit{log}[i][\mathit{index}].\mathit{term} \rangle : \mathit{index} \in 1..\mathit{committedIndex}[i]\} \subseteq \\ \qquad \quad \mathit{committed}(\mathit{currentTerm}[i]) \end{array}$$

Sketch. A leader only advances its committedIndex to cover entries that are immediately committed or prefix committed. Followers update committedIndex from the leader's only when they have a prefix of a prior version of the leader's log.

Proof.

1. The set of committed entries monotonically increases, by its definition.

2. When $\mathit{committedIndex}[i]$ increases, it covers entries present in i's log that are committed:

   (a) Case: follower completes accepting AppendEntries request

      i. Upon processing the request, the follower's log is a prefix of a prior version of the leader's log, l.

      ii. Every entry up to $\mathit{commitIndex}'[i]$ in l is committed by the inductive hypothesis.

   (b) Case: leader i processes AppendEntries response

      i. If the leader sets a new $\mathit{commitIndex}$, the conditions in the spec ensure that $\mathit{commitIndex}'[i] \in \mathit{immediatelyCommitted}$.

      ii. Every entry with index below $\mathit{commitIndex}'[i]$ is prefix committed at $\mathit{currentTerm}[i]$.

3. No log entry below $\mathit{committedIndex}[i]$ is ever removed:

   (a) The leader who sent i the *AppendEntries* request to mark this log prefix committed must have the log prefix (see cases above), and it will not remove any of these entries during its term (Lemma 3).

   (b) By the inductive hypothesis, every leader with a larger term than $\mathit{currentTerm}[i]$ must contain the entry. (And requests from leaders of smaller terms are rejected.)

   (c) If a leader has the entry, it wouldn't have the follower remove it (Lemma 8).

□