MODULE *raft*

This is the formal specification for the Raft consensus algorithm.
Modified by *Ovidiu Marcu*. Simplified model and performance invariants added.

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*

CONSTANTS *MaxClientRequests*

Maximum times a server can become a leader
CONSTANTS *MaxBecomeLeader*

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*

maximum client requests so far
VARIABLE *maxc*

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*

Counter for how many times each server has become leader
VARIABLE *leaderCount*

Maximum term number allowed in the model
CONSTANTS *MaxTerm*

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

The server's term number.
VARIABLE *currentTerm*
The server's state (Follower, *Candidate*, or *Leader*).
VARIABLE *state*
The candidate the server voted for in its current term, or
Nil if it hasn't voted for any.
VARIABLE *votedFor*
$serverVars \triangleq \langle currentTerm, state, votedFor \rangle$

A Sequence of *log* entries. The index into this sequence is the index of the *log* entry. Unfortunately, the Sequence module defines $Head(s)$ as the entry with index 1, so be careful not to use that!
VARIABLE *log*
The index of the latest entry in the *log* the state machine may apply.
VARIABLE *commitIndex*
$logVars \triangleq \langle log, commitIndex \rangle$

The following variables are used only on candidates:
The set of servers from which the candidate has received a *RequestVote* response in its *currentTerm*.
VARIABLE *votesResponded*
The set of servers from which the candidate has received a vote in its *currentTerm*.
VARIABLE *votesGranted*
A history variable used in the proof. This would not be present in an implementation.
Function from each server that voted for this candidate in its *currentTerm* to that voter's *log*.
VARIABLE *voterLog*
$candidateVars \triangleq \langle votesResponded, votesGranted, voterLog \rangle$

The following variables are used only on leaders:

The next entry to send to each follower.

VARIABLE $nextIndex$

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

VARIABLE $matchIndex$

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

End of per server variables.

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

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

Helpers

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

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

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

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

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

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

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

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

$WithMessage(m, msgs) \triangleq$
  IF $m \in$ DOMAIN $msgs$ THEN
    $msgs$
    $[msgs \text{ EXCEPT } ![m] = \text{IF } msgs[m] < 2 \text{ THEN } msgs[m] + 1 \text{ ELSE } 2\ ]$
  ELSE
    $msgs \,@@\, (m :> 1)$

$WithoutMessage(m, msgs) \triangleq$

```
    IF m ∈ DOMAIN msgs THEN
        [msgs EXCEPT ![m] = IF msgs[m] > 0 THEN msgs[m] - 1 ELSE 0]
     ELSE
        msgs
```

Add a message to the bag of messages.

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

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

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

Combination of *Send* and *Discard*

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

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

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

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

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

Define initial values for all variables

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

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

$$\begin{aligned}
InitLeaderVars &\triangleq \land\, nextIndex\ \ = [i \in Server \mapsto [j \in Server \mapsto 1]] \\
&\quad\ \land\, matchIndex = [i \in Server \mapsto [j \in Server \mapsto 0]] \\
InitLogVars &\triangleq \land\, log\ \ \ \ \ \ \ \ \ = [i \in Server \mapsto \langle\rangle] \\
&\quad\ \land\, commitIndex\ = [i \in Server \mapsto 0]
\end{aligned}$$

$$\begin{aligned}
Init \triangleq\ & \land\, messages = [m \in \{\} \mapsto 0] \\
& \land\, InitHistoryVars \\
& \land\, InitServerVars \\
& \land\, InitCandidateVars \\
& \land\, InitLeaderVars \\
& \land\, InitLogVars \\
& \land\, maxc = 0 \\
& \land\, leaderCount = [i \in Server \mapsto 0]
\end{aligned}$$

Define state transitions

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

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

Server i times out and starts a new election.

$$
\begin{aligned}
&Timeout(i) \triangleq \ \land state[i] \in \{Follower, Candidate\} \\
&\qquad\qquad\ \land currentTerm[i] < MaxTerm \\
&\qquad\qquad\ \land state' = [state \text{ EXCEPT } ![i] = Candidate] \\
&\qquad\qquad\ \land currentTerm' = [currentTerm \text{ EXCEPT } ![i] = currentTerm[i] + 1] \\
&\qquad\qquad\ \text{Most implementations would probably just set the local vote} \\
&\qquad\qquad\ \text{atomically, but messaging localhost for it is weaker.} \\
&\qquad\qquad\ \land votedFor' = [votedFor \text{ EXCEPT } ![i] = Nil] \\
&\qquad\qquad\ \land votesResponded' = [votesResponded \text{ EXCEPT } ![i] = \{\}] \\
&\qquad\qquad\ \land votesGranted' = [votesGranted \text{ EXCEPT } ![i] = \{\}] \\
&\qquad\qquad\ \land voterLog' = [voterLog \text{ EXCEPT } ![i] = [j \in \{\} \mapsto \langle\rangle]] \\
&\qquad\qquad\ \land \text{UNCHANGED } \langle messages, leaderVars, logVars, maxc, leaderCount\rangle
\end{aligned}
$$

Candidate i sends j a *RequestVote* request.

$$
\begin{aligned}
&RequestVote(i, j) \triangleq \\
&\quad \land state[i] = Candidate \\
&\quad \land j \notin votesResponded[i] \\
&\quad \land Send([mtype \mapsto RequestVoteRequest, \\
&\qquad\quad mterm \mapsto currentTerm[i], \\
&\qquad\quad mlastLogTerm \mapsto LastTerm(log[i]), \\
&\qquad\quad mlastLogIndex \mapsto Len(log[i]), \\
&\qquad\quad msource \mapsto i, \\
&\qquad\quad mdest \mapsto j]) \\
&\quad \land \text{UNCHANGED } \langle serverVars, candidateVars, leaderVars, logVars, maxc, leaderCount\rangle
\end{aligned}
$$

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

$$
\begin{aligned}
&AppendEntries(i, j) \triangleq \\
&\quad \land i \neq j \\
&\quad \land state[i] = Leader
\end{aligned}
$$

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

Candidate i transitions to leader.
$BecomeLeader(i) \triangleq$
$\land\ state[i] = Candidate$
$\land\ votesGranted[i] \in Quorum$
$\land\ leaderCount[i] < MaxBecomeLeader$
$\land\ state' = [state$ EXCEPT $![i] = Leader]$
$\land\ nextIndex' = [nextIndex$ EXCEPT $![i] =$
$[j \in Server \mapsto Len(log[i]) + 1]]$
$\land\ matchIndex' = [matchIndex$ EXCEPT $![i] =$
$[j \in Server \mapsto 0]]$
$\land\ elections' = elections \cup$
$\{[eterm \mapsto currentTerm[i],$
$eleader \mapsto i,$
$elog \mapsto log[i],$
$evotes \mapsto votesGranted[i],$
$evoterLog \mapsto voterLog[i]]\}$
$\land\ leaderCount' = [leaderCount$ EXCEPT $![i] = leaderCount[i] + 1]$
$\land$ UNCHANGED $\langle messages,\ currentTerm,\ votedFor,\ candidateVars,\ logVars,\ maxc\rangle$

Leader i receives a client request to add v to the *log*.
$ClientRequest(i,\ v) \triangleq$
$\land\ state[i] = Leader$
$\land$ LET $entry \triangleq [term \mapsto currentTerm[i],$
$value \mapsto v]$

```
        newLog ≜ Append(log[i], entry)
    IN  log' = [log EXCEPT ![i] = newLog]
  ∧ UNCHANGED ⟨messages, serverVars, candidateVars,
               leaderVars, commitIndex⟩

ValidMessage(msgs) ≜
    {m ∈ DOMAIN messages : msgs[m] > 0}

ClientRequest(i, v) ≜
    ∧ state[i] = Leader
    ∧ maxc < MaxClientRequests
    ∧ LET entry ≜ [term  ↦ currentTerm[i],
                   value ↦ v]
          entryExists ≜
        ∃ s ∈ Server :
            ∃ j ∈ DOMAIN log[s] :
              log[s][j].value = v
          ∃ j ∈ DOMAIN log[i] : log[i][j].value = v ∧ log[i][j].term = currentTerm[i]
          newLog ≜ IF entryExists
                   THEN log[i]   Keep log unchanged if entry exists
                   ELSE Append(log[i], entry)   Otherwise append the new entry
      IN
        ∧ log' = [log EXCEPT ![i] = newLog]
        ∧ maxc' = IF entryExists THEN maxc ELSE maxc + 1
    ∧ UNCHANGED ⟨messages, serverVars, candidateVars,
                 leaderVars, commitIndex, leaderCount⟩

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 $commitIndex' = [commitIndex \text{ EXCEPT } ![i] = newCommitIndex]$
$\land$ UNCHANGED $\langle messages, serverVars, candidateVars, leaderVars, log, maxc, leaderCount\rangle$

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

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

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

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

$HandleRequestVoteResponse(i, j, m) \triangleq$
This tallies votes even when the current state is not *Candidate*, but they won't be looked at, so it doesn't matter.
$\land m.mterm = currentTerm[i]$
$\land votesResponded' = [votesResponded \text{ EXCEPT } ![i] =$
$votesResponded[i] \cup \{j\}]$
$\land \lor \land m.mvoteGranted$
$\land votesGranted' = [votesGranted \text{ EXCEPT } ![i] =$
$votesGranted[i] \cup \{j\}]$
$\land voterLog' = [voterLog \text{ EXCEPT } ![i] =$
$voterLog[i] \mathbin{@@} (j :> m.mlog)]$
$\lor \land \neg m.mvoteGranted$
$\land$ UNCHANGED $\langle votesGranted, voterLog\rangle$

$\land\ Discard(m)$
$\land$ UNCHANGED $\langle serverVars,\ votedFor,\ leaderVars,\ logVars,\ maxc,\ leaderCount\rangle$

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

$HandleAppendEntriesRequest(i,\ j,\ m) \triangleq$
LET $logOk \triangleq \lor\ m.mprevLogIndex = 0$
$\lor\ \land\ m.mprevLogIndex > 0$
$\land\ m.mprevLogIndex \leq Len(log[i])$
$\land\ m.mprevLogTerm = log[i][m.mprevLogIndex].term$
IN $\land\ m.mterm \leq currentTerm[i]$
$\land\ \lor\ \land$ reject request
$\lor\ m.mterm < currentTerm[i]$
$\lor\ \land\ m.mterm = currentTerm[i]$
$\land\ state[i] = Follower$
$\land\ \neg logOk$
$\land\ Reply([mtype \mapsto AppendEntriesResponse,$
$mterm \mapsto currentTerm[i],$
$msuccess \mapsto$ FALSE,
$mmatchIndex \mapsto 0,$
$msource \mapsto i,$
$mdest \mapsto j],$
$m)$
$\land$ UNCHANGED $\langle serverVars,\ logVars\rangle$
$\lor$ return to follower state
$\land\ m.mterm = currentTerm[i]$
$\land\ state[i] = Candidate$
$\land\ state' = [state$ EXCEPT $![i] = Follower]$
$\land$ UNCHANGED $\langle currentTerm,\ votedFor,\ logVars,\ messages\rangle$
$\lor$ accept request
$\land\ m.mterm = currentTerm[i]$
$\land\ state[i] = Follower$
$\land\ logOk$
$\land$ LET $index \triangleq m.mprevLogIndex + 1$
IN $\lor$ already done with request
$\land\ \lor\ m.mentries = \langle\rangle$
$\lor\ \land\ m.mentries \neq \langle\rangle$
$\land\ Len(log[i]) \geq index$
$\land\ log[i][index].term = m.mentries[1].term$
This could make our *commitIndex* decrease (for example if we process an old, duplicated request), but that doesn't really affect anything.
$\land\ commitIndex' = [commitIndex$ EXCEPT $![i] =$

```
                                    IF commitIndex[i] < m.mcommitIndex THEN
                                        Min({m.mcommitIndex, Len(log[i])})
                                     ELSE
                                        commitIndex[i]]
                    ∧ Reply([mtype          ↦ AppendEntriesResponse,
                             mterm          ↦ currentTerm[i],
                             msuccess       ↦ TRUE,
                             mmatchIndex    ↦ m.mprevLogIndex +
                                              Len(m.mentries),
                             msource        ↦ i,
                             mdest          ↦ j],
                            m)
                    ∧ UNCHANGED ⟨serverVars, log⟩
                ∨ 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, maxc, leaderCount⟩

 Server i receives an AppendEntries response from server j with
 m.mterm = currentTerm[i].
HandleAppendEntriesResponse(i, j, m) ≜
    ∧ m.mterm = currentTerm[i]
    ∧ ∨ ∧ m.msuccess  successful
        ∧ nextIndex'  = [nextIndex  EXCEPT ![i][j]  = IF matchIndex[i][j] > m.mmatchIndex THEN matchI
        ∧ matchIndex' = [matchIndex EXCEPT ![i][j] = IF matchIndex[i][j] > m.mmatchIndex THEN matchI
      ∨ ∧ ¬m.msuccess  not successful
        ∧ nextIndex' = [nextIndex EXCEPT ![i][j] =
                           Max({nextIndex[i][j] − 1, 1})]
        ∧ UNCHANGED ⟨matchIndex⟩
    ∧ Discard(m)
    ∧ UNCHANGED ⟨serverVars, candidateVars, logVars, elections, maxc, leaderCount⟩

 Any RPC with a newer term causes the recipient to advance its term first.
UpdateTerm(i, j, m) ≜
```

```
    ∧ m.mterm > currentTerm[i]
    ∧ m.mterm ≤ MaxTerm
    ∧ 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, maxc, leaderCount⟩

  Responses with stale terms are ignored.
DropStaleResponse(i, j, m) ≜
    ∧ m.mterm < currentTerm[i]
    ∧ Discard(m)
    ∧ UNCHANGED ⟨serverVars, candidateVars, leaderVars, logVars, maxc, leaderCount⟩

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

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

  Network state transitions

  The network duplicates a message
DuplicateMessage(m) ≜
    ∧ Send(m)
    ∧ UNCHANGED ⟨serverVars, candidateVars, leaderVars, logVars, maxc, leaderCount⟩

  The network drops a message
DropMessage(m) ≜
    ∧ Discard(m)
    ∧ UNCHANGED ⟨serverVars, candidateVars, leaderVars, logVars, maxc, leaderCount⟩

──────────────────────────────────────────────────────────────────────
```

Defines how the variables may transition.

$$
\begin{aligned}
Next \triangleq\ & \land \lor \exists\, i \in Server : Timeout(i) \\
& \quad \lor \exists\, i \in Server : Restart(i) \\
& \quad \lor \exists\, i, j \in Server : i \neq j \land RequestVote(i, j) \\
& \quad \lor \exists\, i \in Server : BecomeLeader(i) \\
& \quad \lor \exists\, i \in Server,\ v \in Value : ClientRequest(i, v) \\
& \quad \lor \exists\, i \in Server : AdvanceCommitIndex(i) \\
& \quad \lor \exists\, i, j \in Server : i \neq j \land AppendEntries(i, j) \\
& \quad \lor \exists\, m \in \{msg \in ValidMessage(messages) : \\
& \qquad msg.mtype \in \{RequestVoteRequest,\ RequestVoteResponse,\ AppendEntriesRequest,\ AppendE \\
& \quad \lor \exists\, m \in \{msg \in ValidMessage(messages) : \\
& \qquad msg.mtype \in \{AppendEntriesRequest\}\} : DuplicateMessage(m) \\
& \quad \lor \exists\, m \in \{msg \in ValidMessage(messages) : \\
& \qquad msg.mtype \in \{RequestVoteRequest\}\} : DropMessage(m) \\
& \text{History variable that tracks every } log \text{ ever:} \\
& \land allLogs' = allLogs \cup \{log[i] : i \in Server\}
\end{aligned}
$$

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

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

$$
\begin{aligned}
& MoreThanOneLeaderInv \triangleq \\
& \quad \forall\, i, j \in Server : \\
& \qquad (\land currentTerm[i] = currentTerm[j] \\
& \qquad\ \land state[i] = Leader \\
& \qquad\ \land state[j] = Leader) \\
& \qquad \Rightarrow i = j
\end{aligned}
$$

$$min(a, b) \triangleq \text{IF } a < b \text{ THEN } a \text{ ELSE } b$$

Every (index, term) pair determines a *log* prefix.

From page 8 of the Raft paper: "If two logs contain an entry with the same index and term, then the logs are identical in all pr

$$
\begin{aligned}
& LogMatchingInv \triangleq \\
& \quad \forall\, i, j \in Server : i \neq j \Rightarrow \\
& \qquad \forall\, n \in 1 \,..\, min(Len(log[i]),\ Len(log[j])) : \\
& \qquad\ log[i][n].term = log[j][n].term \Rightarrow \\
& \qquad\ SubSeq(log[i],\ 1,\ n) = SubSeq(log[j],\ 1,\ n)
\end{aligned}
$$

The prefix of the *log* of server i that has been committed up to term x

$$
\begin{aligned}
& CommittedTermPrefix(i, x) \triangleq \\
& \quad \text{Only if } log \text{ of } i \text{ is non-empty, and if there exists an entry up to the term } x \\
& \quad \text{IF } Len(log[i]) \neq 0 \land \exists\, y \in \text{DOMAIN } log[i] : log[i][y].term \leq x \\
& \quad \text{THEN} \\
& \qquad \text{then, we use the subsequence up to the maximum committed term of the leader} \\
& \qquad \text{LET } maxTermIndex \triangleq \\
& \qquad\quad \text{CHOOSE } y \in \text{DOMAIN } log[i] :
\end{aligned}
$$

```
                ∧ log[i][y].term ≤ x
                ∧ ∀ z ∈ DOMAIN log[i] : log[i][z].term ≤ x  ⇒ y ≥ z
        IN   SubSeq(log[i], 1, min(maxTermIndex, commitIndex[i]))
        Otherwise the prefix is the empty tuple
        ELSE  ⟨⟩

CheckIsPrefix(seq1, seq2) ≜
    ∧ Len(seq1) ≤ Len(seq2)
    ∧ ∀ i ∈ 1 .. Len(seq1) : seq1[i] = seq2[i]

The committed entries in every log are a prefix of the
leader's log up to the leader's term (since a next Leader may already be
elected without the old leader stepping down yet)
LeaderCompletenessInv ≜
    ∀ i ∈ Server :
        state[i] = Leader ⇒
        ∀ j ∈ Server : i ≠ j ⇒
            CheckIsPrefix(CommittedTermPrefix(j, currentTerm[i]), log[i])

The prefix of the log of server i that has been committed
Committed(i) ≜
    IF commitIndex[i] = 0
     THEN ⟨⟩
     ELSE SubSeq(log[i], 1, commitIndex[i])

Committed log entries should never conflict between servers
LogInv ≜
    ∀ i, j ∈ Server :
        ∨ CheckIsPrefix(Committed(i), Committed(j))
        ∨ CheckIsPrefix(Committed(j), Committed(i))

Note that LogInv checks for safety violations across space
This is a key safety invariant and should always be checked
THEOREM Spec ⇒ □LogInv

A leader's maxc should remain under MaxClientRequests
MaxCInv ≜ (∃ i ∈ Server : state[i] = Leader) ⇒ maxc ≤ MaxClientRequests

No server can become leader more than MaxBecomeLeader times
LeaderCountInv ≜ ∃ i ∈ Server : (state[i] = Leader ⇒ leaderCount[i] ≤ MaxBecomeLeader)

No server can have a term exceeding MaxTerm
MaxTermInv ≜ ∀ i ∈ Server : currentTerm[i] ≤ MaxTerm

MyConstraint ≜ (∀ i ∈ Server : currentTerm[i] ≤ 3 ∧ Len(log[i]) ≤ 3) ∧ (∀ m ∈ DOMAIN messages : messages

Symmetry ≜ Permutations(Server)
```