# 1 Case Study on Raft

We divide the specification of the Raft protocol into three modules: module PreVote describing the pre-vote mechanism, module Vote describing the election mechanism and module Replication describing the transmission of log entries from the leader to the followers. For each module, we mainly discuss how the interaction-preserving abstraction is conducted, as detailed below.

## 1.1 Abstraction for Module Replication

In Raft, log entries are replicated from leader to follower in sequence. A follower accepts a log entry from leader only if it has already accepted all previous log entries. If it receives a log entry when preceding entries are not fully accepted, it rejects the entry. When an entry is rejected, leader tries to send the previous one in its log. As leader does not know exactly which entry the follower would accept in an asynchronous distributed system, the process of sending and rejecting may take multiple rounds before the follower can accept its missing entry, in which the state variables of the leader and follower remain unchanged. Any newly elected leader has to find each follower's first unmatched log entry by such process. Therefore, a trace containing multiple elections can be very long due to such "invalid" communications between leader and followers. Also many system states are generated due to the uncertain order between these invalid actions and other actions.

Figure 1 lists three actions of module Replication. Action *AppenEntries* specifies the process when leader $s_1$ sends an *AppendEntries* request to some follower $s_2$ to replicate log entries within the cluster. This action only modifies internal variable *net* which records all messages sent by servers. When a follower receives a log entry from leader, it performs the prefix check to ensure that it has already received all previous log entries. If prefix check fails, the follower simply responds to leader with the index of unmatched entry without modifying any other variable, as is specified by action $ReplicateFailUnmatch$. If a follower receives from leader exactly the log entry it misses, it adds the entry to its log and sends back an ack, as is specified by *Replicate*. Figure 2 lists actions specifying how leader handles responses from followers. When leader receives a response from a follower indicating that the entry is accepted, it records the match index and the index of next log entry to send to the follower, as is specified by $HandleAppendEntriesResponseSuccess$. Note that $matchIndex$ and $nextIndex$ are specific to replication mechanism implementation and thus internal variables which can be omitted in the abstracted specification. When a follower rejects the log entry from leader, leader learns that this log entry is not the first one the follower misses. So it reduces $nextIndex$ for the follower by 1 and tries to replicate the last

$$
\begin{array}{l}
AppendEntries(s1,\ s2) \triangleq \text{LET } ind \triangleq nextIndex[s1][s2] \\
\quad \text{IN} \\
\quad\quad \land\ state[s1] = \text{“LEADER”} \\
\quad\quad \land\ s1 \neq s2 \\
\quad\quad \land\ Send(net, \\
\quad\quad\quad\quad [mtype\ \mapsto \text{“AppendEntries”}, \\
\quad\quad\quad\quad\ mterm \mapsto currentTerm[s1], \\
\quad\quad\quad\quad\ mprevTerm \mapsto log[s1][ind-1].term, \\
\quad\quad\quad\quad\ mprevIndex \mapsto ind-1, \\
\quad\quad\quad\quad\ mentry \mapsto e, \\
\quad\quad\quad\quad\ msrc \mapsto s1, \\
\quad\quad\quad\quad\ mdest \mapsto s2])
\end{array}
$$

$$
\begin{array}{l}
ReplicateFailUnmatch(s1,\ s2) \triangleq \exists\, m \in net : \\
\quad\quad \land\ m.mtype = \text{“AppendEntries”} \\
\quad\quad \land\ m.msrc = s1 \\
\quad\quad \land\ m.mdest = s2 \\
\quad\quad \land\ \lor\ Len(log[s2]) < m.mprevIndex \\
\quad\quad\quad\ \lor\ \land\ Len(log[s2]) \geq m.mprevIndex \\
\quad\quad\quad\quad\ \land\ log[s2][m.mprevIndex].term \neq m.mprevTerm \\
\quad\quad \land\ Send(net, \\
\quad\quad\quad\quad [mtype\ \mapsto \text{“AppendEntriesResponse”}, \\
\quad\quad\quad\quad\ mterm \mapsto currentTerm'[s2], \\
\quad\quad\quad\quad\ msuccess \mapsto \text{FALSE}, \\
\quad\quad\quad\quad\ mindex \mapsto m.mprevIndex+1, \\
\quad\quad\quad\quad\ msrc \mapsto s2, \\
\quad\quad\quad\quad\ mdest \mapsto m.msrc])
\end{array}
$$

$$
\begin{array}{l}
Replicate(s1,\ s2) \triangleq \exists\, m \in net : \\
\quad\quad \land\ m.mtype = \text{“AppendEntries”} \\
\quad\quad \land\ m.mdest = s2 \\
\quad\quad \land\ m.msrc = s1 \\
\quad\quad \land\ \lor\ m.mprevIndex = 0 \\
\quad\quad\quad\ \lor\ \land\ m.mprevIndex \neq 0 \\
\quad\quad\quad\quad\ \land\ Len(log[s2]) \quad \geq m.mprevIndex \\
\quad\quad\quad\quad\ \land\ log[s2][m.mprevIndex].term = m.mprevTerm \\
\quad\quad \land\ log'[s2] = Append(SubSeq(log[s2],\ 1,\ m.mprevIndex),\ m.mentry) \\
\quad\quad \land\ Send(net, \\
\quad\quad\quad\quad [mtype\ \mapsto \text{“AppendEntriesResponse”}, \\
\quad\quad\quad\quad\ mterm \mapsto currentTerm'[s2], \\
\quad\quad\quad\quad\ msuccess \mapsto \text{TRUE}, \\
\quad\quad\quad\quad\ mindex \mapsto m.mprevIndex+1, \\
\quad\quad\quad\quad\ msrc \mapsto s2, \\
\quad\quad\quad\quad\ mdest \mapsto m.msrc])
\end{array}
$$

Figure 1: Specification for Replication

$$
\begin{aligned}
HandleAppendEntriesResponseFail(s) \triangleq\ & \exists\, m \in net : \\
& \land\ m.mtype\ =\ \text{“AppendEntriesResponse”} \\
& \land\ m.mdest\ =\ s \\
& \land\ m.mterm = currentTerm[s] \\
& \land\ state[s] = \text{“LEADER”} \\
& \land\ m.msuccess = \text{FALSE} \\
& \land\ nextIndex'[s][m.msrc] = max(1,\ nextIndex[s][m.msrc] - 1)
\end{aligned}
$$

$$
\begin{aligned}
HandleAppendEntriesResponseSuccess(s) \triangleq\ & \exists\, m \in net : \\
& \land\ m.mtype = \text{“AppendEntriesResponse”} \\
& \land\ m.mdest = s \\
& \land\ state[s] = \text{“LEADER”} \\
& \land\ m.mterm = currentTerm[s] \\
& \land\ m.msuccess = \text{TRUE} \\
& \land\ m.mindex > matchIndex[s][m.msrc] \\
& \land\ nextIndex'[s][m.msrc] = m.mindex + 1 \\
& \land\ matchIndex'[s][m.msrc] = m.mindex
\end{aligned}
$$

Figure 2: Specification for Replication (continued)

log entry. Action $HandleAppendEntriesResponseFail$ specifies this process.

$$
\begin{aligned}
Replicate(s1,\ s2) \triangleq\ & \text{LET}\ t \triangleq currentTerm[s2]\ \text{IN} \\
& \land\ leader[t] = s1 \\
& \land\ s1 \neq s2 \\
& \land\ \exists\, j \in 1\,..\,Len(leaderLog[t]) : \\
& \quad \land\ consistent(leaderLog[t],\ log[s2],\ j - 1) \\
& \quad \land\ \lor\ Len(log[s2]) < j \\
& \qquad \lor\ \land\ Len(log[s2]) \geq j \\
& \qquad\quad \land\ leaderLog[t][j].term \neq log[s2][j].term \\
& \quad \land\ log'[s2] = Append(SubSeq(log[s2],\ 1,\ j - 1),\ en)
\end{aligned}
$$

Figure 3: Abstracted Specification for Replication

To conduct abstraction, a leader simply sends the exact log entry that followers miss without the process of trial and error because TLA+ allows users to model a distributed system in terms of a single global state and a leader can utilize global state of each server. Thus, the redundant steps of sending and receiving “invalid” messages in system traces are eliminated. Figure 3 shows the single action of the abstracted specification for this process. This action corresponds to action *Replicate* in Figure 1. All other actions and internal variables such as $nextIndex$ and $matchIndex$ are reduced by abstraction.

## 1.2 Abstraction for Module PreVote

Raft relies on a leader election algorithm to elect a single leader for each term. If follower does not receive heart beat messages from leader for some time, it becomes candidate and

starts an election by increasing its term and sending election messages concurrently to other system servers. Since network may be unreliable, a server partitioned from leader cannot receive messages from leader. Therefore, it tries to start election for multiple times and increases its term to a large value. When network condition becomes normal, its large term would be propagated within the cluster, forcing the leader to step down and the cluster has to elect a new leader unnecessarily.

To prevent such occasional network fluctuation from causing disruptions, Raft introduces the pre-vote mechanism. When a follower tries to start an election, it has to send pre-vote requests to other servers. Servers grant or refuse pre-vote requests based on their system states. Only if the server learns from a majority of the cluster that they would grant its pre-vote request can it increases its term and make election proposals. Pre-vote mechanism solves the issue of partitioned server disrupting the cluster when it rejoins since a partitioned server cannot increase its term unless a majority of the cluster agree to elect a new leader.

Figure 4 shows the three actions of module PreVote. The behavior that a follower starts an election by changing its state to $PreCandidate$ and sending election requests to other servers for network reasons is modeled by action $PreVote$. Servers handle received election requests by executing action $HandlePreVote$. If the sender receives granted responses from a majority of servers, it changes state to $Candidate$ and updates its term, as is specified by action $BecomeCandidate$.

Note that according to action $HandlePreVote$, servers handle pre-vote requests by sending response messages without modifying their system states, which means that the action is transparent to other modules. We can thus omit this action in the abstraction of module PreVote. Figure 5 is the abstracted version containing 2 actions, each corresponding to $PreVote$ and $BecomeCandidate$ respectively. In action $AbsPreVote$, except for changing follower's state, a history variable $preVoteSet$ is used to record all possible servers that may grant the follower's pre-vote request. The follower trying to start an election can change state to $Candidate$ only if its $preVoteSet$ contains a quorum of servers, as is specified by action $AbsPreVote$. In this way, the action of handling election request is omitted while preserving the functionality of pre-vote mechanism.

## 1.3 Abstraction for Module Vote

Raft's leader election algorithm can also be abstracted similarly. To become a leader, a candidate sends election request currently to all servers requesting for votes. A server may grant or refuse a vote request according to its state and the information the message contains. Only when the candidate's vote is approved by a majority of servers can it become leader. Thus, a round of election takes multiple steps in a behavior trace as servers work

$$
\begin{aligned}
PreVote(s) \triangleq\ & \land\ state[s] = \text{“FOLLOWER”} \\
& \land\ state'[s] = \text{“PRE\_CANDIDATE”} \\
& \land\ Send(net, [mtype \mapsto \text{“PreVoteRequest”}, \\
& \qquad mterm \mapsto currentTerm[s], \\
& \qquad msrc \mapsto s, \\
& \qquad mlog \mapsto log[s]])
\end{aligned}
$$

$$
\begin{aligned}
HandlePreVote(s) \triangleq\ & \land \exists\, m \in net : \\
& \qquad \land\ m.mtype = \text{“PreVoteRequest”} \\
& \qquad \land\ m.mterm = currentTerm[s] \\
& \qquad \land\ \text{LET } granted \triangleq CompareLog(log[s],\ m.mlog) \\
& \qquad\ \ \text{IN} \\
& \qquad\qquad Send(net, [mtype \mapsto \text{“PreVoteResponse”}, \\
& \qquad\qquad\qquad mterm \mapsto currentTerm[s], \\
& \qquad\qquad\qquad msrc \mapsto s, \\
& \qquad\qquad\qquad mdest \mapsto m.msrc, \\
& \qquad\qquad\qquad mgranted \mapsto granted])
\end{aligned}
$$

$$
\begin{aligned}
BecomeCandidate(s) \triangleq\ & \land\ state[s] = \text{“PRE\_CANDIDATE”} \\
& \land \exists\, Q \in Quorums : \\
& \qquad \forall\, q \in Q : \exists\, m \in net : \\
& \qquad\qquad \land\ m.mtype = \text{“PreVoteResponse”} \\
& \qquad\qquad \land\ m.mterm = currentTerm[s] \\
& \qquad\qquad \land\ m.mdest = s \\
& \qquad\qquad \land\ m.msrc = q \\
& \qquad\qquad \land\ m.mgranted = \text{TRUE} \\
& \land\ state'[s] = \text{“CANDIDATE”} \\
& \land\ currentTerm'[s] = currentTerm[s] + 1
\end{aligned}
$$

Figure 4: Specification for PreVote

$$
\begin{aligned}
AbsPreVote \triangleq\ & \land\ state[s] = \text{“FOLLOWER”} \\
& \land\ preVoteSet'[s] = \{a \in Server : canVote(s,\ a) = \text{TRUE}\} \\
& \land\ state'[s] = \text{“PRE\_CANDIDATE”}
\end{aligned}
$$

$$
\begin{aligned}
AbsBecomeCandidate(s) \triangleq\ & \land\ state[s] = \text{“PRE\_CANDIDATE”} \\
& \land\ preVoteSet[s] \in Quorum \\
& \land\ state' = [state \text{ EXCEPT } ![s] = \text{“CANDIDATE”}] \\
& \land\ currentTerm'[s] = currentTerm[s] + 1
\end{aligned}
$$

Figure 5: Abstracted Specification for PreVote

asynchronously and each server's handling of the vote request takes one step. More over, as network is unreliable and messages can be delayed arbitrarily, the order each server handles the election request is undetermined, adding much more system states to be checked.

A natural abstraction for election is to choose a server and change its role to leader, taking only one step and avoiding possible permutations due to asynchrony. However, the election algorithm is delicately designed to ensure that every newly elected leader meets several essential properties such as single leader and leader completeness, which are critical to the correctness of Raft. Therefore, we figure out and specify these properties in our specification. With this, we can specify that an eligible server change its state to leader as abstraction for election, omitting details while perfectly matching original design.

Figure 6 shows the four actions of module Election. Action $RequestVote$ specifies a candidate sends election requests to all other servers when starting a new election. This action only changes variable $net$, which records all messages sent by servers. $net$ is an internal variable, so action $RequestVote$ can be omitted by abstraction. Servers may grant or refuse election requests by comparing candidate's $term$ and $log$ with their own. Action $MakeVoteFailLowTerm$ specifies the case when a server refuses an election request because the candidate's $term$ is smaller. Action $MakeVoteFailOldLog$ specifies the case when the candidate's election request is refused because of outdated log. Both these two actions change no variables except for $net$ and thus can also be omitted. If the candidate's $term$ and $log$ are no older than the follower who received candidate's election reuqest, it grants the election request by sending an ack, as is specified by action $MakeVote$. This action also only modifies variable $net$. If a candidate receives ack from a quorum of servers, it changes its state to $LEADER$, as action $BecomeLeader$ specifies.

Figure 7 shows the abstracted specification of leader election. It contains only one action, $BecomeLeader$, which is enabled only if the two essential properties $singleLeader$ and $leaderCompleteness$, are true. Thus, any successful leader election in the abstracted specification guarantee these two properties. By such abstraction, a round of election takes only 2 steps no matter how many servers are in the cluster, greatly reducing the complexity of election algorithm, especially when the number of servers is big.

$$
\begin{aligned}
RequestVote(s) \triangleq\ & \land\ state[s] = \text{“CANDIDATE”} \\
& \land\ Send(net, [mtype \mapsto \text{“RequestVote”}, \\
& \qquad mterm \mapsto currentTerm[s], \\
& \qquad mlogState \mapsto logState(s), \\
& \qquad msrc \mapsto s])
\end{aligned}
$$

$$
\begin{aligned}
MakeVoteFailLowTerm(s) \triangleq\ & \land \exists\, m \in net : \\
& \qquad \land\ m.mtype = \text{“RequestVote”} \\
& \qquad \land\ m.mterm < currentTerm[s] \\
& \qquad \land\ Send(net, [mtype \mapsto \text{“RequestVoteResponse”}, \\
& \qquad\qquad mterm \mapsto currentTerm[s], \\
& \qquad\qquad mgranted \mapsto \text{FALSE}, \\
& \qquad\qquad msrc \mapsto s, \\
& \qquad\qquad mdest \mapsto m.msrc])
\end{aligned}
$$

$$
\begin{aligned}
MakeVoteFailOldLog(s) \triangleq\ & \land \exists\, m \in net : \\
& \qquad \land\ m.mtype = \text{“RequestVote”} \\
& \qquad \land\ m.mterm = currentTerm[s] \\
& \qquad \land\ compareLog(log[s],\ m.mlogState) = \text{FALSE} \\
& \qquad \land\ Send(net, [mtype \mapsto \text{“RequestVoteResponse”}, \\
& \qquad\qquad mterm \mapsto currentTerm[s], \\
& \qquad\qquad mgranted \mapsto \text{FALSE}, \\
& \qquad\qquad msrc \mapsto s, \\
& \qquad\qquad mdest \mapsto m.msrc])
\end{aligned}
$$

$$
\begin{aligned}
MakeVote(s) \triangleq\ & \land \exists\, m \in net : \\
& \qquad \land\ m.mtype = \text{“RequestVote”} \\
& \qquad \land\ m.mterm = currentTerm[s] \\
& \qquad \land\ compareImp(log[s],\ m.mlogState) \\
& \qquad \land\ Send(net, [mtype \mapsto \text{“RequestVoteResponse”}, \\
& \qquad\qquad mterm \mapsto currentTerm[s], \\
& \qquad\qquad mgranted \mapsto \text{TRUE}, \\
& \qquad\qquad msrc \mapsto s, \\
& \qquad\qquad mdest \mapsto m.msrc])
\end{aligned}
$$

$$
\begin{aligned}
BecomeLeader(s) \triangleq\ & \land \exists\, Q \in Quorums : \\
& \qquad \forall\, q \in Q : \exists\, m \in net : \\
& \qquad\qquad \land\ m.mtype = \text{“RequestVoteResponse”} \\
& \qquad\qquad \land\ m.mgranted = \text{TRUE} \\
& \qquad\qquad \land\ m.msrc = q \\
& \qquad\qquad \land\ m.mterm = currentTerm[s] \\
& \qquad\qquad \land\ m.mdest = s \\
& \qquad \land\ state[s] = \text{“CANDIDATE”} \\
& \qquad \land\ state'[s] = \text{“LEADER”}
\end{aligned}
$$

Figure 6: Specification for Vote

$$
\begin{aligned}
BecomeLeader(s) \triangleq\ & \wedge state[s] = \text{“CANDIDATE”} \\
& \wedge singleLeader(s,\ currentTerm[s]) = \text{TRUE} \\
& \wedge leaderCompleteness(s,\ currentTerm[s]) = \text{TRUE} \\
& \wedge state'[s] = \text{“LEADER”}
\end{aligned}
$$

Figure 7: Abstracted Specification for Vote

# 2 Case Study on ParallelRaft

PRaft specification is divided into two modules: module LeaderRecovery and module FollowerRecovery. We conduct interaction-preserving abstraction on PRaft mainly from three perspectives, as detailed below.

## 2.1 Asynchrony Elimination

The centralized coordinator learns the states of servers through polling. Nodes respond to polling messages by sending replies with their system states. In implementation level specification, such polling process takes multiple steps to finish as there are several nodes in the cluster and nodes handle coordinator message asynchronously, each node responding to the polling message takes one step. When performing leader election, coordinator has to learn the checkpoint of each server. We found that the checkpoint of each server stays unchanged during polling as no valid leader exists and client commands cannot be replicated among the cluster. This suggests that the polling process is transparent to other modules. Therefore, in abstracted specification, the coordinator learns the states of all nodes synchronously in abstraction, which is safe and takes only one step. By such abstraction, traces with different permutations of polling message handling actions are all mapped to a same trace of abstracted specification.

Figure 8 shows the three actions of polling process. When performing leader recovery, coordinator sends to servers requesting checkpoint of each server by action *GetLogLen*. Server handle this request by action *HandleGeyLogLen*. When coordinator receives responses from a majority of servers, it does local calculation using replies, as is specified by Action *ProcessServerLog*. We found that when state of coordinator is *REC_LEADER*, checkpoints remain unchanged, which suggests that the order of servers handling “LogLenRequest” is irrelevant.

Figure 9 shows abstracted specification. It has a single action in which coordinator directly do local calculation using globally available checkpoints of servers. This is a typical example showing how we do abstraction for one module by eliminating asynchronous

$$
\begin{array}{l}
GetLogLen \triangleq \land\ ctrlState = \text{“REC\_LEADER”} \\
\qquad\qquad\ \land\ Send(net, [mtype \mapsto \text{“LogLenRequest”}])
\end{array}
$$

$$
\begin{array}{l}
HandleGetLogLen(s) \triangleq \land \exists\, m \in net : \\
\qquad\qquad\qquad\qquad\ \land\ mmtype = \text{“LogLenRequest”} \\
\qquad\qquad\qquad\qquad\ \land\ Send(net, [mtype \mapsto \text{“LogLenResponse”}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ msrc \mapsto s, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ mlogLen \mapsto len(log[s])]
\end{array}
$$

$$
\begin{array}{l}
ProcessServerLog \triangleq \land\ ctrlState = \text{“REC\_LEADER”} \\
\qquad\qquad\qquad\ \land \exists\, Q \in Quorums : \\
\qquad\qquad\qquad\qquad \land \forall\, s \in Q : \\
\qquad\qquad\qquad\qquad\quad \exists\, m \in net : \land\ m.mtype = \text{“LogLenResponse”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ m.msrc = s \\
\qquad\qquad\qquad\qquad \land\ DoSelection(Q)
\end{array}
$$

Figure 8: Specification for Selection

$$
\begin{array}{l}
DirectProcessServerLog \triangleq \land\ ctrlState = \text{“REC\_LEADER”} \\
\qquad\qquad\qquad\qquad\ \land \exists\, Q \in Quorum : \\
\qquad\qquad\qquad\qquad\quad DoSelection(Q)
\end{array}
$$

Figure 9: Abstracted Specification for Selection

behaviors that are transparent to other modules.

## 2.2 Control Flow Simplification

$$
\begin{aligned}
ProgressRequest(s) \triangleq\ & \land\ ctrlState = \text{“WAIT\_TASK1”} \\
& \land\ Send([net,\ [mtype \mapsto \text{“ProgressRequest”},\\
& \qquad\qquad mdest \mapsto s]
\end{aligned}
$$

$$
\begin{aligned}
ProgressResponse(s) \triangleq\ & \land \exists\, m \in net : \\
& \qquad \land\ m.mtype = \text{“ProgressRequest”} \\
& \qquad \land\ m.mdest = s \\
& \qquad \land\ Send([net,\ mtype \mapsto \text{“ProgressResponse”},\\
& \qquad\qquad\qquad m.msrc \mapsto s,\\
& \qquad\qquad\qquad mfinished \mapsto Task1Finished(s)]
\end{aligned}
$$

$$
\begin{aligned}
Task1Finished \triangleq\ & \land\ ctrlState = \text{“WAIT\_TASK1”} \\
& \land \exists\, m \in net : \\
& \quad \land\ m.mtype = \text{“ProgressResponse”} \\
& \quad \land\ m.mfinished = \text{TRUE} \\
& \quad \land\ Send(net,\ [mtype \mapsto \text{“DoTask2”},\\
& \qquad\qquad mdest \mapsto m.msrc])
\end{aligned}
$$

$$
\begin{aligned}
BeginTask2(s) \triangleq\ & \land \exists\, m \in net : \\
& \quad \land\ m.mtype = \text{“DoTask2”} \\
& \quad \land\ m.mdest = s \\
& \quad \land\ DoTask2(s)
\end{aligned}
$$

Figure 10: Specification for a Typical Control Flow

Figure 10 shows a typical control flow of PRaft. Coorinator periodically checks whether a server has finished some task it assigns as is specified by action *ProgressRequest*. When the coordinator learns from the server that it has finished, coordinator sends a message requesting the server to begin doing subsequent task. Action $Task1Finished$ specifies this process. When receiving request from coordinator, server executes command as ordered.

Figure 11 shows the abstracted specification for this process. It omit the first three actions. When a server finishes one task, it starts doing subsequent task autonomously, as if it received an order from the coordinator. Also coordinator knows the progress of each server using global information. Therefore, the abstracted specification allows the same system behaviors as the implementation-level specification.

$$
\begin{aligned}
AbsBeginTask2(s) \triangleq\ & \wedge\ Task1Finished(s) = \text{TRUE} \\
& \wedge\ Task2Finished(s) = \text{FALSE} \\
& \wedge\ DoTask2(s)
\end{aligned}
$$

Figure 11: Abstracted Specification for Control Flow

## 2.3 Omit Implementation Optimizations

Figure 12 shows the specification of coordinator collecting logs from a majority of servers using two rounds of communication. In the first round, it simply learns the length of each server's log. The first two actions specify this process. When coordinator receives replies from a majority of servers, it chooses the ones with longer logs and requests their logs, as is specified by action *RequestLog*. Note only the selected servers can receive this request. When receiving request from coordinator, server sends back its log, as is specified by action *SendLog*. Note that whether a server send coordinator the whole log or just its length makes no difference in model checking since network capacity is not considered.

Network capacity is not modeled in specification, so it has no effect on the cost of model checking. But more rounds of communication introduce more steps in behavior traces and more possible permutations of actions, which increase the cost of model checking. Thus, we choose the solution with one round of communication in the abstracted specification.

Figure 13 is the abstracted specification. Each server simply sends the coordinator its whole log directly, taking only one round of communication. Note logs that coordinator receives from servers are internal variable of module leader recovery, so this difference has no influence on the other modules.

$$
\begin{aligned}
RequestLogFreshness \triangleq\ & \land\ ctrlState = \text{“REC\_LEADER”} \\
& \land\ Send(net, [mtype \mapsto \text{“LogFreshnessRequest”}])
\end{aligned}
$$

$$
\begin{aligned}
RequestLogResponse(s) \triangleq\ & \land \exists\, m \in net : \\
& \quad \land\ m.mtype = \text{“LogFreshnessRequest”} \\
& \quad \land\ Send(net, [mtype \mapsto \text{“LogFreshnessResponse”}, \\
& \qquad\qquad msrc \mapsto s, \\
& \qquad\qquad mlogLen \mapsto Len(log[s])])
\end{aligned}
$$

$$
\begin{aligned}
RequestLog \triangleq\ & \land \exists\, Q \in Quorums : \\
& \quad \land \forall\, q \in Q : \\
& \qquad \exists\, m \in net : m.mtype = \text{“LogFreshnessResponse”} \\
& \quad \land \text{LET } selected \triangleq SelectLog(Q) \\
& \quad \text{IN} \\
& \qquad Send(net, [mtype \mapsto \text{“LogRequest”}, \\
& \qquad\qquad mdest \mapsto selected])
\end{aligned}
$$

$$
\begin{aligned}
SendLog(s) \triangleq\ & \land \exists\, m \in net : \\
& \quad \land\ m.mtype = \text{“LogRequest”} \\
& \quad \land\ m.mdest = s \\
& \land\ Send(net, [mtype \mapsto \text{“LogResponse”}, \\
& \qquad\qquad msrc \mapsto s, \\
& \qquad\qquad mlog \mapsto log[s])
\end{aligned}
$$

Figure 12: Specification Containing Two Rounds of Communication

$$
\begin{aligned}
AbsRequestLogResponse(s) \triangleq\ & Send(net, [mtype \mapsto \text{“LogResponse”}, \\
& \qquad\qquad msrc \mapsto sm \\
& \qquad\qquad mlog \mapsto log[s]])
\end{aligned}
$$

Figure 13: Abstracted Specification Containing One Round of Communication