Hovercraft++ TLA+ Specification

Ovidiu-Cristian Marcu[1*]

[1*]Computer Science, University of Luxembourg, Luxembourg.

Corresponding author(s). E-mail(s): ovidiu21marcu@gmail.com;

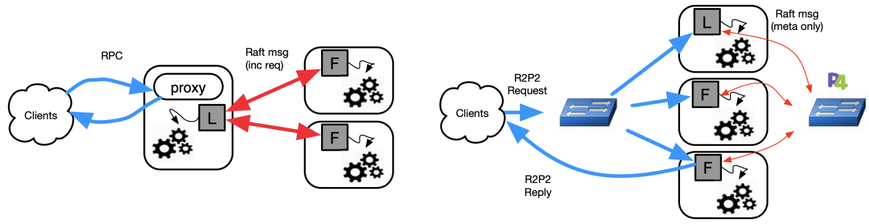
Abstract

We introduce a formal TLA+ specification of the Hovercraft++ consensus protocol, derived from its natural language description in *HovercRaft: Achieving Scalability and Fault-tolerance for microsecond-scale Datacenter Services.* While the original description provides a strong foundation, our TLA+ model facilitates a deeper, rigorous analysis. We focus on the core consensus mechanism's fundamental safety properties, while excluding load balancing components (client replies, read-only operations) and bounded queue optimizations. Model checking the specification allows us to uncover and verify behavior in corner cases not overtly addressed in the original paper. Furthermore, this formalization process itself clarifies the intricacies of Raft-based consensus (upon which our specification, like the original TLA+ Raft specification it extends, is built), offering insights beyond even what standard Raft specifications typically detail. This experience reinforces our recommendation for researchers to invest in formal protocol specification with detailed commentary, as it significantly clarifies contributions, aids potential implementations, and ultimately strengthens protocol design.

Keywords: consensus, hovercraft++, TLA+, specification

1 Background: the Hovercraft++ Consensus Protocol

Hovercraft++ [1], *"an approach by which adding nodes increases both the resilience and the performance of general-purpose state-machine replication (SMR)"*, is an extension of the Raft [2] consensus protocol, designed to enhance the efficiency and scalability of SMR. SMR and its bottlenecks are described in subsections 2.1.1 and 2.1.2 of [1]. This paper investigates how Hovercraft++ preserves Raft's core safety while liveness guarantees investigation is left for future work. For the purpose of formal specification, its key provisions are:



(a) Scalability bottlenecks of Raft. **(b)** HovercRaft, with optional in-network acceleration using a P4 ASIC.

Fig. 1 "Eliminating bottlenecks of SMR. Figure 1a shows the leader node bottlenecks for a classic SMR deployment using Raft: (1) the leader acts as the RPC server for all clients; (2) the leader must communicate individually with each follower to replicate messages and ensure their ordering. Figure 1b illustrates Hovercraft++, extending Raft to separate request replication from ordering and using IP multicast (Switch) and in-network accelerators (NetAgg) to convert leader-to-multipoint interactions into point-to-point interactions. Illustration on a 3-node cluster" reproduced from [1].

- **Transparent SMR Integration:** It integrates an extension of Raft directly within the R2P2 [3] transport layer, enabling applications to utilize SMR without modification to their core logic for handling consensus. "Specifically, the SMR layer becomes part of the RPC layer which forwards RPC requests to the application layer only after those requests have been totally ordered and committed by the leader" [1]. Design details provided in Section 3 of [1].
- **Modified Replication and Ordering Mechanics:** Client request (including payload) replication is separated from the leader's ordering task by using IP multicast for request dissemination to all nodes through a Switch. The leader is still responsible on establishing the total order of these received requests. For specific design details we refer the reader to Section 3.2 of [1].
- **In-Network Acceleration of Core SMR Messaging:** Hovercraft++ leverages in-network programmable hardware (e.g., P4 ASICs) to statelessly manage crucial SMR communication patterns. This includes:
  - The fan-out of leader messages (like AppendEntries containing ordering metadata) to followers.
  - The fan-in of follower replies related to these consensus messages.

  The in-network aggregator offloading is viewed as a leader extension and it **aims** to improve the efficiency of the consensus protocol's internal communication, particularly as cluster size increases, without altering the fundamental Raft algorithm for achieving agreement. However, the Hovercraft++ protocol was not formally specified nor verified. For specific design details we refer the reader to Sections 4 (Figure 5.b explains communication in Hovercraft++), and 5 of [1]; subsection 6.4 describes important details about the NetAgg aggregator implementation of which we implement the AGG_COMMIT message to ensure Raft servers update their commit index to be able to respond to clients.

Our TLA+ [1] specification of the Hovercraft++ protocol is based on its design presented in Sections 3, 4, and 5 [1]. We postpone load balancing client replies, load

[1] Lamport's TLA+ home page: https://lamport.azurewebsites.net/tla/tla.html

balancing read-only operations and bounded queues (sections 3.3 to 3.6 and partially section 5 of [1]) to future work discussing liveness and client guarantees. Our specification extends the original Raft TLA+ specification [2] to additionally model the Switch and NetAgg components introduced in Hovercraft++, as illustrated in Figure 1.

2 Hovercraft TLA+ Specification

[2] Raft TLA+ specification https://github.com/ongardie/raft.tla

MODULE *hovercraftpp*

* Formal specification of the *HoverCraft* ++ consensus algorithm.
* See paper: https://*marioskogias.github.io*/docs/*hovercraft.pdf*
* Based on the Raft consensus algorithm with enhancements for improved throughput.
* Original Raft TLA+ specification: https://*github.com*/ongardie/*raft.tla*
* See also the Raft paper: https://*raft.github.io*/*raft.pdf*
*
* This specification introduces two key components :
* − *Switch*: Abstracts multicast/broadcast mechanisms to decouple *payload* replication
* from consensus ordering, reducing leader bandwidth bottlenecks
* − *NetAgg*: Network aggregator that collects acknowledgments from *followers*
* and sends commit notifications, further reducing leader coordination overhead
*
* We verify safety properties and track message counts for commit index advancement.
* The specification supports server crashes and leader election mechanisms.
*
* Modified by *Ovidiu Marcu*.

In standard Raft the leader is the central hub. A client sends a request only to the leader. The leader must then replicate the entire request *payload* to all followers. The leader gathers acknowledgments from Followers, commits the entry, applies it, and sends a response to the client. Bottleneck: The leader's network bandwidth and processing power for sending the full *payload* to every follower becomes a limiting factor as the cluster size (N) or client request size increases. Throughput is limited by $Leader\_Bandwidth \ / \ ((N-1) * Request\_Payload\_Size)$.

We introduce a "Switch" abstraction (representing mechanisms like *IP* Multicast or a dedicated middlebox/programmable switch as used in the *HovercRaft* paper). Clients send requests via *Switch*. The *Switch* is non-crashing. Client requests timeouts due to *Switch* failures will force clients choose another *Switch*. The *Switch*'s responsibility is to deliver the request *payload* to all server nodes (Leader and Followers) "simultaneously". The 'simultaneous' delivery is a model abstraction for efficient broadcast mechanisms like *IP* Multicast, where *payload* dissemination is handled by the network infrastructure. We introduce the "*NetAgg*" network aggregation component of *HovercRaft* ++ . The *NetAgg* is non-crashing and stateless. Client requests will timeout and retry. The leader sends a single message containing the ordering metadata to *NetAgg*. Leader expects a *AggCommit* message in return. If one is not received in due time, the *Leader* declares *NetAgg* failure, another *NetAgg* will be chosen. *NetAgg* is then responsible for disseminating this metadata to followers and collecting their acknowledgments. When a follower receives the ordering metadata message from *NetAgg*, it uses the identifier in the metadata to find the corresponding *payload* in its temporary buffer. Once matched, the follower places the request *payload* into its replicated *log* at the correct index specified by the leader's metadata. Once *NetAgg* receives acknowledgments from a quorum of followers, it sends an *AggCommit* message to all servers, informing them that they can advance their commit index for that entry. *AggCommit* also ensures *NetAgg* failure handling by *Leader* choosing another *NetAgg*. This model assumes that the *Switch* and *NetAgg* components will never fail. We do not model *Switch* and *NetAgg* replicas due to model space constraints. Point-to-point recovery mechanisms between Followers and the *Leader* are partially addressed (assumes no *Follower* crashes).

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

************************* CONSTANTS ***

The set of server *IDs* including the *Switch* and *NetAgg*

CONSTANTS *Server*

The set of client requests that can go into the *log*

Represents the possible values of client requests that are stored in the *log*.
A value with its term represent the request metadata. Same value is used as *payload*. A typical flow for a request's data/metadata might be :

* Client $\rightarrow$ *Switch* (metadata and *payload*)
* *Switch* $\rightarrow$ *Leader* & Followers (metadata and *payload*)
* *Leader* $\rightarrow$ *NetAgg* (metadata for ordering)
* *NetAgg* $\rightarrow$ Followers $\rightarrow$ *NetAgg* (metadata for ordering and acknowledgements)
* *NetAgg* $\rightarrow$ All *Servers* (*AggCommit* sent when majority quorum of acks received)
* One server of *Servers* will reply back to client (not handled by this model).

CONSTANTS *Value*

Server states for Raft protocol.

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

Fixed states for *Switch* and *NetAgg* indices in *Server*

CONSTANTS *Switch*, *NetAgg*

A reserved value.

CONSTANTS *Nil*

Message types

Standard Raft message types for leader election and *log* replication.

CONSTANTS *RequestVoteRequest*, *RequestVoteResponse*,
AppendEntriesRequest, *AppendEntriesResponse*

Hovercraft ++ specific message types for interactions involving the *NetAgg* component.

CONSTANTS *AppendEntriesNetAggRequest*, *AggCommit*

Limits the total number of client requests processed in the model,
used for bounding the state space during model checking.

CONSTANTS *MaxClientRequests*

Limits the number of times any given server can transition to
the *Leader* state, for state space bounding.

CONSTANTS *MaxBecomeLeader*

Defines the maximum value a term number can reach, used for state space bounding.

CONSTANTS *MaxTerm*

************************ VARIABLES **

Global variables

A bag of records representing requests and responses sent from one server
to another. This is a function mapping Message to *Nat*.

VARIABLE $messages$

An instrumentation variable mapping each server ID (from $Server$)
to a natural number, counting how many times it has become a leader.
Used with $MaxBecomeLeader$ for state space bounding.

VARIABLE $leaderCount$

An instrumentation variable; a natural number tracking the count of
client requests processed so far. Used with $MaxClientRequests$ for
state space bounding.

VARIABLE $maxc$

variable for tracking entry commit message counts
Maps $\langle logIndex, logTerm \rangle$ to a record tracking message counts.
[$sentCount \mapsto Nat$, \ * $AppendEntriesRequests$ sent for the entry
$ackCount \mapsto Nat$, \ * $AppendEntriesResponses$ received for this entry
$committed \mapsto Bool$] \ * Flag indicating if the entry is committed

VARIABLE $entryCommitStats$

A tuple grouping all instrumentation-specific variables. Useful for specifying
UNCHANGED $instrumentationVars$ in actions that do not modify them.

$$instrumentationVars \triangleq \langle leaderCount, maxc, entryCommitStats \rangle$$

The unique identifier (ID from $Server$ set) of the server designated
to act as the $Switch$ component.

VARIABLE $switchIndex$

The $Switch$'s internal buffer. Maps a request identifier ($\langle value, term \rangle$)
to the complete request data and $payload$

VARIABLE $switchBuffer$

A per-server variable (maps $Server$ ID to a set of request identifiers).
For each server (Leader and Followers), it stores the set of request
identifiers ($\langle value, term \rangle$ tuples) for payloads received from the $Switch$
that are awaiting ordering metadata.

VARIABLE $unorderedRequests$

Records which $\langle value, term \rangle$ pairs the $Switch$ has sent to each server.
Maps $Server$ $ID \rightarrow$ Set of $\langle Value, Term \rangle$ pairs.

VARIABLE $switchSentRecord$

A tuple grouping variables specific to the $Hovercraft$ $Switch$ functionality.

$$hovercraftVars \triangleq \langle switchBuffer, unorderedRequests, switchIndex, switchSentRecord \rangle$$

$NetAgg$ variables

Stores the leader (leader field, type $Server$) and its term (term field, type Nat)

that *NetAgg* currently recognizes as active.

VARIABLE $netAggCurrentLeaderTerm$

VARIABLE $netAggIndex$ Index of the *NetAgg* server

VARIABLE $netAggMatchIndex$ *NetAgg*'s view of follower match indices

VARIABLE $netAggPendingEntries$ Entries pending aggregation at *NetAgg*

VARIABLE $netAggCommitIndex$ *NetAgg*'s view of commit index

A tuple grouping all variables specific to the *NetAgg* component.

$$netAggVars \triangleq \langle netAggIndex, netAggMatchIndex, netAggPendingEntries, netAggCommitIndex, netAggCurrentLeaderTerm \rangle$$

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

Each server's current known term number. (Maps *Server ID* to *Nat*).

VARIABLE $currentTerm$

The server's state (Follower, *Candidate*, *Leader*, *Switch*, or *NetAgg*).
For *Switch* and *NetAgg* entities, this state is fixed.

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.

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 $matchIndex$

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

The set of server *IDs* participating in the Raft consensus ($i.e.$, excluding *Switch* and *NetAgg* components).
$Servers \triangleq Server \setminus \{switchIndex, netAggIndex\} \setminus *$ see *Init*
VARIABLE $Servers$

All variables; used for stuttering (asserting state hasn't changed).
Hovercraft ++ brings *hovercraftVars* for *Switch* and *netAggVars* for *NetAgg*.
$vars \triangleq \langle messages, serverVars, candidateVars, leaderVars, logVars,$
$\quad instrumentationVars, hovercraftVars, netAggVars, Servers \rangle$

*********************** HELPERS ***

Defines the set of all possible quorums. A quorum is any subset of *Servers* (Raft participants) forming a simple majority. The critical property is that any two quorums must overlap.
$Quorum \triangleq \{i \in \text{SUBSET } (Servers) : Cardinality(i) * 2 > Cardinality(Servers)\}$

The term of the last entry in a *log*, or 0 if the *log* is empty.
$LastTerm(xlog) \triangleq \text{IF } Len(xlog) = 0 \text{ THEN } 0 \text{ ELSE } xlog[Len(xlog)].term$

$WithMessage(m, msgs) \triangleq$
$\quad \text{IF } m \in \text{DOMAIN } msgs \text{ THEN}$
$\quad\quad msgs$ avoiding duplicates
$\quad \text{ELSE}$
$\quad\quad msgs \mathbin{@@} (m :> 1)$

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

$WithoutMessage(m, msgs) \triangleq$
$\quad \text{IF } m \in \text{DOMAIN } msgs \text{ THEN}$
$\quad\quad [msgs \text{ EXCEPT } ![m] = \text{IF } msgs[m] > 0 \text{ THEN } msgs[m] - 1 \text{ ELSE } 0]$
$\quad \text{ELSE}$
$\quad\quad msgs$

Add a message to the bag of messages.

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

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

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

Helper for *Send* and *Reply*. Given a message m and bag of messages, return a Combination of *Send* and *Discard*

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

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

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

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

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

Convert a sequence to a set of its elements

$SeqToSet(seq) \triangleq \{seq[i] : i \in \text{DOMAIN } seq\}$

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

$$
\begin{aligned}
&ValidMessage(msgs) \triangleq \\
&\quad \{m \in \text{DOMAIN } messages : msgs[m] > 0\}
\end{aligned}
$$

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

$CommittedTermPrefix(i, x) \triangleq$

Only if *log* of i is non-empty, and if there exists an entry up to the term x

$\quad \text{IF } Len(log[i]) \neq 0 \land \exists y \in \text{DOMAIN } log[i] : log[i][y].term \leq x$

$\quad \text{THEN}$

then, we use the subsequence up to the maximum committed term of the leader

$$
\begin{aligned}
&\quad\quad \text{LET } maxTermIndex \triangleq \\
&\quad\quad\quad\quad \text{CHOOSE } y \in \text{DOMAIN } log[i] : \\
&\quad\quad\quad\quad\quad \land\, log[i][y].term \leq x \\
&\quad\quad\quad\quad\quad \land\, \forall z \in \text{DOMAIN } log[i] : log[i][z].term \leq x \Rightarrow y \geq z \\
&\quad\quad \text{IN } \quad SubSeq(log[i], 1, min(maxTermIndex, commitIndex[i]))
\end{aligned}
$$

Otherwise the prefix is the empty tuple

$\quad \text{ELSE } \langle\rangle$

$$
\begin{aligned}
&CheckIsPrefix(seq1, seq2) \triangleq \\
&\quad \land\, Len(seq1) \leq Len(seq2) \\
&\quad \land\, \forall i \in 1 \,..\, Len(seq1) : seq1[i] = seq2[i]
\end{aligned}
$$

The prefix of the *log* of server i that has been committed

$$
\begin{aligned}
&Committed(i) \triangleq \\
&\quad \text{IF } commitIndex[i] = 0 \\
&\quad\ \text{THEN } \langle\rangle
\end{aligned}
$$

$\quad$ ELSE $SubSeq(log[i], 1, commitIndex[i])$

$$MyConstraint \triangleq (\forall i \in Servers : currentTerm[i] \leq MaxTerm \land Len(log[i]) \leq MaxClientRequests) \land (\forall m \in \text{DOMAIN } messages : messages[m] \leq 1)$$

*********************** *INIT* ***

$$InitHistoryVars \triangleq voterLog = [i \in Servers \mapsto [j \in \{\} \mapsto \langle\rangle]]$$

$$InitServerVars \triangleq \land currentTerm = [i \in Servers \mapsto 1] \land state = [i \in Servers \mapsto Follower] \land votedFor = [i \in Servers \mapsto Nil]$$

$$InitCandidateVars \triangleq \land votesResponded = [i \in Servers \mapsto \{\}] \land votesGranted = [i \in Servers \mapsto \{\}]$$

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

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

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

$Init \triangleq$

$\land messages = [m \in \{\} \mapsto 0]$

$\land switchIndex = \text{CHOOSE } s \in Server : \text{TRUE}$ Pick any server as switch

$\land netAggIndex = \text{CHOOSE } n \in Server \setminus \{switchIndex\} : \text{TRUE}$ Pick another as $NetAgg$

$\land Servers = Server \setminus \{switchIndex, netAggIndex\}$ Remaining are Raft servers

Initialize all server state

$\land currentTerm = [i \in Server \mapsto 1]$

$\land state = [i \in Server \mapsto$ IF $i = switchIndex$ THEN $Switch$ ELSE IF $i = netAggIndex$ THEN $NetAgg$ ELSE $Follower]$

$\land votedFor = [i \in Server \mapsto Nil]$

Initialize empty logs and indices

$\land log = [i \in Server \mapsto \langle\rangle]$

$\land commitIndex = [i \in Server \mapsto 0]$

$\land nextIndex = [i \in Server \mapsto [j \in Server \mapsto 1]]$

$\land matchIndex = [i \in Server \mapsto [j \in Server \mapsto 0]]$

Initialize candidate variables

$\land votesResponded = [i \in Server \mapsto \{\}]$

$\land$ *votesGranted* $= [i \in Server \mapsto \{\}]$
$\land$ *voterLog* $= [i \in Server \mapsto [j \in \{\} \mapsto \langle\rangle]]$

Initialize *HoverCraft Switch* variables
$\land$ *switchBuffer* $= [vt \in \{\} \mapsto \{\}]$
$\land$ *unorderedRequests* $= [s \in Server \mapsto \{\}]$
$\land$ *switchSentRecord* $= [s \in Server \mapsto \{\}]$

Initialize *HoverCraft NetAgg* variables
$\land$ *netAggCurrentLeaderTerm* $= Nil$
$\land$ *netAggMatchIndex* $= [s \in \{\} \mapsto 0]$
$\land$ *netAggPendingEntries* $= \{\}$
$\land$ *netAggCommitIndex* $= 0$

Initialize instrumentation
$\land$ *maxc* $= 0$
$\land$ *leaderCount* $= [i \in Server \mapsto 0]$
$\land$ *entryCommitStats* $= [idx\_term \in \{\} \mapsto$
$\quad [sentCount \mapsto 0,\ ackCount \mapsto 0,\ committed \mapsto \text{FALSE}]]$

Used to start from a state with a *Leader*.
We may verify just the normal case excluding leader election and crashes.

$MyInit \triangleq$
$\quad \text{LET } ServerSet5 \triangleq \text{CHOOSE } S \in \text{SUBSET } (Server) : Cardinality(S) = 5$
$\quad\quad TheSwitchId \triangleq \text{CHOOSE } s \in ServerSet5 : \text{TRUE}$
$\quad\quad TempSet \triangleq ServerSet5 \setminus \{TheSwitchId\}$
$\quad\quad TheNetAggId \triangleq \text{CHOOSE } n \in TempSet : \text{TRUE}$
$\quad\quad TempSet2 \triangleq TempSet \setminus \{TheNetAggId\}$
$\quad\quad TheLeaderId \triangleq \text{CHOOSE } l \in TempSet2 : \text{TRUE}$
$\quad\quad FollowerIds \triangleq TempSet2 \setminus \{TheLeaderId\}$

$\quad\quad TheState \triangleq [s \in Server \mapsto$
$\quad\quad\quad \text{IF } s = TheSwitchId \text{ THEN } Switch$
$\quad\quad\quad \text{ELSE IF } s = TheNetAggId \text{ THEN } NetAgg$
$\quad\quad\quad \text{ELSE IF } s = TheLeaderId \text{ THEN } Leader$
$\quad\quad\quad \text{ELSE IF } s \in FollowerIds \text{ THEN } Follower$
$\quad\quad\quad \text{ELSE } Follower$
$\quad\quad]$
$\quad\quad TheSwitchIndex \triangleq TheSwitchId$
$\quad\quad TheNetAggIndex \triangleq TheNetAggId$
$\quad\quad TheServersSet \triangleq Server \setminus \{TheSwitchIndex, TheNetAggIndex\}$
$\quad\quad Voters \triangleq TheServersSet \setminus \{TheLeaderId\}$
$\quad \text{IN}$

Constraint: Ensure *Server* has enough elements
$\land\ Cardinality(Server) \geq 5$
$\land\ PrintT($"MyInit: switchIndex=" $\circ\ ToString(TheSwitchIndex))$

$\wedge\ PrintT(\text{“MyInit: netAggIndex=”} \circ ToString(TheNetAggIndex))$
$\wedge\ PrintT(\text{“MyInit: Leader is=”} \circ ToString(TheLeaderId))$
$\wedge\ PrintT(\text{“MyInit: Servers=”} \circ ToString(TheServersSet))$
$\wedge\ PrintT(\text{“}MyInit : state[switchIndex] = \text{”} \circ ToString(TheState[TheSwitchIndex]))$
$\wedge\ PrintT(\text{“}MyInit : state[LeaderId] = \text{”} \circ ToString(TheState[TheLeaderId]))$
$\wedge\ PrintT(\text{“}MyInit : switchBuffer\ Domain = \text{”} \circ ToString(\text{DOMAIN}\ [vt \in \{\} \mapsto \{\}]))$

$\wedge\ commitIndex = [s \in Server \mapsto 0]$
$\wedge\ currentTerm = [s \in Server \mapsto 2]$
$\wedge\ leaderCount = [s \in Server \mapsto \text{IF}\ s = TheLeaderId\ \text{THEN}\ 1\ \text{ELSE}\ 0]$
$\wedge\ log = [s \in Server \mapsto \langle\rangle]$
$\wedge\ matchIndex = [s \in Server \mapsto [t \in Server \mapsto 0]]$
$\wedge\ maxc = 0$
$\wedge\ messages = [m \in \{\} \mapsto 0]$
$\wedge\ nextIndex = [s \in Server \mapsto [t \in Server \mapsto 1]]$
$\wedge\ state = TheState$
$\wedge\ votedFor = [s \in Server \mapsto$
$\quad \text{IF}\ s = TheLeaderId\ \text{THEN}\ Nil\ \text{ELSE}\ TheLeaderId]$
$\wedge\ voterLog = [s \in Server \mapsto$
$\quad \text{IF}\ s = TheLeaderId\ \text{THEN}$
$\quad [v \in Voters \mapsto \langle\rangle]\ \text{ELSE}\ [v \in \{\} \mapsto \langle\rangle]]$
$\wedge\ votesGranted = [s \in Server \mapsto$
$\quad \text{IF}\ s = TheLeaderId\ \text{THEN}\ Voters\ \text{ELSE}\ \{\}]$
$\wedge\ votesResponded = [s \in Server \mapsto$
$\quad \text{IF}\ s = TheLeaderId\ \text{THEN}\ Voters\ \text{ELSE}\ \{\}]$
$\wedge\ entryCommitStats = [idx\_term \in \{\} \mapsto$
$\quad [sentCount \mapsto 0,$
$\quad ackCount \mapsto 0,$
$\quad committed \mapsto \text{FALSE}]]$
$\wedge\ switchBuffer = [vt \in \{\} \mapsto \{\}]$
$\wedge\ unorderedRequests = [s \in Server \mapsto \{\}]$
$\wedge\ switchSentRecord = [s \in Server \mapsto \{\}]$
$\wedge\ switchIndex = TheSwitchIndex$
$\wedge\ netAggIndex = TheNetAggIndex$
$\wedge\ netAggMatchIndex = [s \in TheServersSet \mapsto 0]$
$\wedge\ netAggPendingEntries = \{\}$
$\wedge\ netAggCommitIndex = 0$
$\wedge\ netAggCurrentLeaderTerm = [leader \mapsto TheLeaderId,\ term \mapsto 2]$
$\wedge\ Servers = TheServersSet$

********************* Actions ******************************

Modified to limit Restarts only for Leaders.
Server i restarts from stable storage.
It loses everything but its *currentTerm*, *votedFor*, and *log*.
Also persists messages, instrumentation and *Switch*/*NetAgg* variables.

$$
\begin{array}{l}
Restart(i) \triangleq \\
\quad \land\ state[i] = Leader \\
\quad \land\ (\forall srv \in Servers : leaderCount[srv] < MaxBecomeLeader) \\
\quad \land\ state' = [state \text{ EXCEPT } ![i] = Follower] \\
\quad \land\ votesResponded' = [votesResponded \text{ EXCEPT } ![i] = \{\}] \\
\quad \land\ votesGranted' = [votesGranted \text{ EXCEPT } ![i] = \{\}] \\
\quad \land\ voterLog' = [voterLog \text{ EXCEPT } ![i] = [j \in \{\} \mapsto \langle\rangle]] \\
\quad \land\ nextIndex' = [nextIndex \text{ EXCEPT } ![i] = [j \in Server \mapsto 1]] \\
\quad \land\ matchIndex' = [matchIndex \text{ EXCEPT } ![i] = [j \in Server \mapsto 0]] \\
\quad \land\ commitIndex' = [commitIndex \text{ EXCEPT } ![i] = 0] \\
\quad \land\ unorderedRequests' = [unorderedRequests \text{ EXCEPT } ![i] = \{\}] \\
\quad \land\ switchSentRecord' = [switchSentRecord \text{ EXCEPT } ![i] = \{\}] \\
\quad \land\ \text{IF } netAggCurrentLeaderTerm \neq Nil \land netAggCurrentLeaderTerm.leader = i \\
\qquad \text{THEN } \land\ netAggCurrentLeaderTerm' = Nil \quad \text{Deactivate } NetAgg \\
\qquad\qquad\quad \land\ netAggPendingEntries' = \{\} \quad \text{Flush pending entries} \\
\qquad\qquad\quad netAggCommitIndex \text{ and } netAggMatchIndex \text{ could be left or reset;} \\
\qquad\qquad\quad \text{they become irrelevant until a new leader activates } NetAgg. \\
\qquad \text{ELSE } \land\ \text{UNCHANGED } netAggCurrentLeaderTerm \\
\qquad\qquad\quad \land\ \text{UNCHANGED } netAggPendingEntries \\
\quad \land\ \text{UNCHANGED } \langle messages, currentTerm, votedFor, log, instrumentationVars, \\
\qquad\qquad switchIndex, switchBuffer, Servers, \\
\qquad\qquad netAggIndex, netAggMatchIndex, netAggCommitIndex\rangle
\end{array}
$$

Server i times out and starts a new election. $Follower \rightarrow Candidate$

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

Modified to restrict $Leader$ transitions, bounded by $MaxBecomeLeader$

Candidate i transitions to leader. $Candidate \rightarrow Leader$

$$
\begin{array}{l}
BecomeLeader(i) \triangleq \\
\quad \land\ state[i] = Candidate \\
\quad \land\ votesGranted[i] \in Quorum \\
\quad \land\ leaderCount[i] < MaxBecomeLeader
\end{array}
$$

$\land$ $state' = [state$ EXCEPT $![i] = Leader]$
$\land$ $nextIndex' = [nextIndex$ EXCEPT $![i] =$
  $[j \in Server \mapsto Len(log[i]) + 1]]$
$\land$ $matchIndex' = [matchIndex$ EXCEPT $![i] =$
  $[j \in Server \mapsto 0]]$
$\land$ $leaderCount' = [leaderCount$ EXCEPT $![i] = leaderCount[i] + 1]$
$\land$ $netAggCurrentLeaderTerm' = [leader \mapsto i, term \mapsto currentTerm[i]]$
$\land$ $netAggPendingEntries' = \{\}$ Flush pending entries for the new leader
$\land$ $netAggCommitIndex' = commitIndex[i]$
$\land$ $netAggMatchIndex' = [s \in Servers \mapsto 0]$
$\land$ UNCHANGED $\langle messages, currentTerm, votedFor, candidateVars,$
  $logVars, maxc, entryCommitStats, hovercraftVars,$
  $Servers, netAggIndex\rangle$

Modified up to *MaxTerm*; Back To *Follower*.
Any *RPC* with a newer term causes the recipient to advance its term first.

$UpdateTerm(i, j, m) \triangleq$
$\land$ $state[i] \notin \{Switch, NetAgg\} \land state[j] \notin \{Switch, NetAgg\}$
$\land$ $m.mterm > currentTerm[i]$
$\land$ $m.mterm < MaxTerm$
$\land$ LET $wasLeader \triangleq state[i] = Leader$
  IN
  $\land$ $currentTerm' = [currentTerm$ EXCEPT $![i] = m.mterm]$
  $\land$ $state' = [state$ EXCEPT $![i] = Follower]$
  $\land$ $votedFor' = [votedFor$ EXCEPT $![i] = Nil]$
  $\land$ IF $wasLeader \land netAggCurrentLeaderTerm \neq Nil \land netAggCurrentLeaderTerm.leader = i$
    THEN $\land$ $netAggCurrentLeaderTerm' = Nil$ Deactivate *NetAgg*
      $\land$ $netAggPendingEntries' = \{\}$ Flush pending entries
    ELSE $\land$ UNCHANGED $netAggCurrentLeaderTerm$
      $\land$ UNCHANGED $netAggPendingEntries$
  messages is unchanged so m can be processed further.
$\land$ UNCHANGED $\langle messages, candidateVars, leaderVars, logVars,$
  $instrumentationVars, hovercraftVars, Servers,$
  $netAggIndex, netAggMatchIndex, netAggCommitIndex\rangle$

*************************** REQUEST VOTE **********************************

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

Candidate i sends j a *RequestVote* request.

$RequestVote(i, j) \triangleq$
$\land$ $state[i] = Candidate$
$\land$ $state[j] \notin \{Switch, NetAgg\}$
$\land$ $j \notin votesResponded[i]$
$\land$ $Send([mtype \mapsto RequestVoteRequest,$

$$
\begin{array}{l}
\qquad\qquad mterm \mapsto currentTerm[i], \\
\qquad\qquad mlastLogTerm \mapsto LastTerm(log[i]), \\
\qquad\qquad mlastLogIndex \mapsto Len(log[i]), \\
\qquad\qquad msource \mapsto i, \\
\qquad\qquad mdest \mapsto j]) \\
\quad \land \textsc{unchanged}\ \langle serverVars, candidateVars, leaderVars, logVars, \\
\qquad\qquad instrumentationVars, hovercraftVars, Servers, netAggVars \rangle
\end{array}
$$

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

$$
\begin{array}{l}
HandleRequestVoteRequest(i, j, m) \triangleq \\
\quad \textsc{let}\ logOk \triangleq \lor m.mlastLogTerm > LastTerm(log[i]) \\
\qquad\qquad\qquad \lor \land m.mlastLogTerm = LastTerm(log[i]) \\
\qquad\qquad\qquad\quad \land m.mlastLogIndex \geq Len(log[i]) \\
\qquad\quad grant \triangleq \land m.mterm = currentTerm[i] \\
\qquad\qquad\qquad \land logOk \\
\qquad\qquad\qquad \land votedFor[i] \in \{Nil, j\} \\
\quad \textsc{in}\ \land m.mterm \leq currentTerm[i] \\
\qquad \land \lor grant \land votedFor' = [votedFor\ \textsc{except}\ ![i] = j] \\
\qquad\quad \lor \neg grant \land \textsc{unchanged}\ votedFor \\
\qquad \land Reply([mtype \mapsto RequestVoteResponse, \\
\qquad\qquad mterm \mapsto currentTerm[i], \\
\qquad\qquad mvoteGranted \mapsto grant,
\end{array}
$$

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

$$
\begin{array}{l}
\qquad\qquad mlog \mapsto log[i], \\
\qquad\qquad msource \mapsto i, \\
\qquad\qquad mdest \mapsto j], \\
\qquad\qquad m) \\
\qquad \land \textsc{unchanged}\ \langle state, currentTerm, candidateVars, leaderVars, logVars, \\
\qquad\qquad instrumentationVars, hovercraftVars, Servers, netAggVars \rangle
\end{array}
$$

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.

$$
\begin{array}{l}
\quad \land m.mterm = currentTerm[i] \\
\quad \land votesResponded' = [votesResponded\ \textsc{except}\ ![i] = \\
\qquad\qquad\qquad votesResponded[i] \cup \{j\}] \\
\quad \land \lor \land m.mvoteGranted \\
\qquad\quad \land votesGranted' = [votesGranted\ \textsc{except}\ ![i] = \\
\qquad\qquad\qquad votesGranted[i] \cup \{j\}] \\
\qquad\quad \land voterLog' = [voterLog\ \textsc{except}\ ![i] = \\
\qquad\qquad\qquad voterLog[i]\ @@\ (j :> m.mlog)]
\end{array}
$$

```
    ∨ ∧ ¬m.mvoteGranted
      ∧ UNCHANGED ⟨votesGranted, voterLog⟩
  ∧ Discard(m)
  ∧ UNCHANGED ⟨serverVars, votedFor, leaderVars, logVars,
               instrumentationVars, hovercraftVars, Servers, netAggVars⟩
```

Responses with stale terms are ignored.

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

*************************** AppendEntries ********************************

Leader i ingests a request v that has been replicated to its unordered set.

```
LeaderIngestHovercRaftRequest(i, vt) ≜
  ∧ state[i] = Leader
  ∧ vt ∈ unorderedRequests[i]       Request ID is pending for the leader
  ∧ vt ∈ DOMAIN switchBuffer        use switch buffer to reduce payload duplication
  ∧ maxc < MaxClientRequests
  ∧ LET entryFromBuffer ≜ switchBuffer[vt]
        v ≜ vt[1]   Extract value from ⟨value, term⟩ pair
        Use leader's current term, keep value and payload from buffer
        newEntry ≜ [term ↦ currentTerm[i],
                    value ↦ v,
                    payload ↦ entryFromBuffer.payload]
        entryExists ≜ ∃ k ∈ DOMAIN log[i] :
                        log[i][k].value = v ∧ log[i][k].term = newEntry.term
        newLog ≜ IF entryExists THEN log[i] ELSE Append(log[i], newEntry)
        newEntryIndex ≜ Len(log[i]) + 1
        newEntryKey ≜ ⟨newEntryIndex, newEntry.term⟩
    IN
      ∧ log' = [log EXCEPT ![i] = newLog]
      ∧ maxc' = IF entryExists THEN maxc ELSE maxc + 1
      ∧ entryCommitStats' =
           IF ¬entryExists ∧ newEntryIndex > 0
            THEN entryCommitStats @@ (newEntryKey :> [sentCount ↦ 0,
                                      ackCount ↦ 0, committed ↦ FALSE])
            ELSE entryCommitStats
      ∧ unorderedRequests' = [unorderedRequests EXCEPT ![i] = @ \ {vt}]
  ∧ UNCHANGED ⟨messages, serverVars, candidateVars, leaderVars,
               commitIndex, leaderCount, switchIndex, switchBuffer,
               Servers, switchSentRecord, netAggVars⟩
```

Client sends a request to the *Switch*, which buffers it,
not yet replicated to servers
s is the *Switch*, *i* is the leader, *v* is the request value
(along with term will represent a request *ID* in this model)

$SwitchClientRequest(s, i, v) \triangleq$
$\quad \land state[s] = Switch$ Only the switch server can process client requests
$\quad \land state[i] = Leader$
$\quad \land$ LET $vt \triangleq \langle v, currentTerm[i] \rangle$ Create $\langle value, term \rangle$ pair
$\quad\quad$ IN
$\quad\quad \land vt \notin$ DOMAIN $switchBuffer$ Only process new requests
$\quad\quad \land$ LET $entryWithPayload \triangleq [term \mapsto currentTerm[i], value \mapsto v, payload \mapsto v]$
$\quad\quad\quad$ IN
$\quad\quad\quad \land switchBuffer' = switchBuffer\ @@\ (vt :> entryWithPayload)$
$\quad\quad\quad \land unorderedRequests' = [unorderedRequests$ EXCEPT $![s] = unorderedRequests[s] \cup \{vt\}]$
$\quad \land$ UNCHANGED $\langle messages, serverVars, candidateVars, leaderVars, logVars, leaderCount, entryCommitStats, switchIndex, maxc, Servers, switchSentRecord, netAggVars \rangle$

check that *Server i* is not a *Switch* or *NetAgg*

$RaftState(i) \triangleq state[i] \notin \{Switch, NetAgg\}$

The *Switch* replicates *vt* to ALL servers at once (except those that already have it).
This reduces state space by avoiding intermediate states
where only some servers have received the request.

$SwitchClientRequestReplicateAll(s, vt) \triangleq$
$\quad \land state[s] = Switch$ Only the switch server can replicate requests
$\quad \land vt \in unorderedRequests[s]$ Request must be pending at the switch
$\quad \land$ LET Find all servers that haven't received this v/term pair yet

$\quad\quad targetServers \triangleq \{i \in Server : RaftState(i) \land vt \notin switchSentRecord[i]\}$
$\quad\quad$ IN
$\quad\quad \land targetServers \neq \{\}$ At least one server needs the request
$\quad\quad \land unorderedRequests' = [i \in Server \mapsto$ IF $i \in targetServers$ THEN $unorderedRequests[i] \cup \{vt\}$ ELSE $unorderedRequests[i]]$
$\quad\quad \land switchSentRecord' = [i \in Server \mapsto$ IF $i \in targetServers$ THEN $switchSentRecord[i] \cup \{vt\}$ ELSE $switchSentRecord[i]]$
$\quad \land$ UNCHANGED $\langle messages, serverVars, candidateVars, leaderVars, logVars, leaderCount, entryCommitStats, switchIndex, switchBuffer, maxc, Servers, netAggVars \rangle$

Follower i drops/loses one request vt from its unordered requests
This simulates network loss or follower crash scenarios

$FollowerDropRequest(i, vt) \triangleq$
$\quad \land\ state[i] = Follower$ Only followers can drop requests
$\quad \land\ vt \in unorderedRequests[i]$ Request must exist in follower's buffer
$\quad \land\ vt \in \textsc{domain}\ switchBuffer$ Request must still exist in switch buffer
$\quad \land\ unorderedRequests' = [unorderedRequests\ \textsc{except}\ ![i] = @ \setminus \{vt\}]$
$\quad \land\ \textsc{unchanged}\ \langle messages, serverVars, candidateVars, leaderVars, logVars,$
$\qquad instrumentationVars, switchIndex, switchBuffer,$
$\qquad switchSentRecord, Servers, netAggVars \rangle$

Leader i sends *AppendEntries* to *NetAgg* instead of directly to followers

$AppendEntriesToNetAgg(i) \triangleq$
$\quad \land\ state[i] = Leader$
$\quad \land\ state[netAggIndex] = NetAgg$
$\quad \land\ Len(log[i]) > 0$
$\quad \land\ \textsc{let}\ nextIndexMin \triangleq Min(\{nextIndex[i][j] : j \in Servers \setminus \{i\}\})$
$\qquad \textsc{in}\ nextIndexMin \leq Len(log[i])$
$\quad \land\ \textsc{let}\ entryIndex \triangleq Min(\{nextIndex[i][j] : j \in Servers \setminus \{i\}\})$
$\qquad\quad entry \triangleq log[i][entryIndex]$
$\qquad\quad entryMetadata \triangleq [term \mapsto entry.term, value \mapsto entry.value]$
$\qquad\quad entries \triangleq \langle entryMetadata \rangle$
$\qquad\quad prevLogIndex \triangleq entryIndex - 1$
$\qquad\quad prevLogTerm \triangleq \textsc{if}\ prevLogIndex > 0\ \textsc{then}$
$\qquad\qquad log[i][prevLogIndex].term$
$\qquad\qquad \textsc{else}\ 0$
$\qquad \textsc{in}\ Send([mtype \mapsto AppendEntriesNetAggRequest,$
$\qquad\qquad mterm \mapsto currentTerm[i],$
$\qquad\qquad mprevLogIndex \mapsto prevLogIndex,$
$\qquad\qquad mprevLogTerm \mapsto prevLogTerm,$
$\qquad\qquad mentries \mapsto entries,$
$\qquad\qquad mentryIndex \mapsto entryIndex,$
$\qquad\qquad mlog \mapsto log[i],$
$\qquad\qquad mcommitIndex \mapsto Min(\{commitIndex[i], entryIndex\}),$
$\qquad\qquad msource \mapsto i,$
$\qquad\qquad mdest \mapsto netAggIndex])$
$\quad \land\ \textsc{unchanged}\ \langle serverVars, candidateVars, leaderVars, logVars,$
$\qquad instrumentationVars, hovercraftVars, netAggVars, Servers, netAggVars \rangle$

NetAgg receives *AppendEntries* from leader and forwards to ALL followers atomically

$NetAggForwardAppendEntriesAll(m) \triangleq$
$\quad \land\ m.mdest = netAggIndex$
$\quad \land\ m.mtype = AppendEntriesNetAggRequest$
$\quad \land\ netAggCurrentLeaderTerm \neq Nil$ *NetAgg* must be active
$\quad \land\ netAggCurrentLeaderTerm.leader = m.msource$ Request from current assigned leader

$\land$ $netAggCurrentLeaderTerm.term = m.mterm$
$\land$ LET $leaderId \triangleq m.msource$
$followers \triangleq Servers \setminus \{leaderId\}$ All servers except the leader
Create the set of messages to send to all followers
$followerMessages \triangleq \{[mtype \mapsto AppendEntriesRequest,$
$mterm \mapsto m.mterm,$
$mprevLogIndex \mapsto m.mprevLogIndex,$
$mprevLogTerm \mapsto m.mprevLogTerm,$
$mentries \mapsto m.mentries,$
$mlog \mapsto m.mlog,$
$mcommitIndex \mapsto m.mcommitIndex,$
$msource \mapsto netAggIndex,$
$mdest \mapsto f,$
$moriginalLeader \mapsto leaderId]$
$: f \in followers\}$
Remove the processed message and add all new messages
$RemainingActiveMessages \triangleq ValidMessage(WithoutMessage(m, messages))$
Update *sentCount* for this entry
$entryIndex \triangleq m.mentryIndex$
$entryTerm \triangleq m.mentries[1].term$
$entryKey \triangleq \langle entryIndex, entryTerm\rangle$
IN
$\land$ $messages' = [msgRec \in RemainingActiveMessages \cup followerMessages \mapsto 1]$
$\land$ $netAggPendingEntries' = netAggPendingEntries \cup$
$\{[entryIndex \mapsto m.mentryIndex,$
$entryTerm \mapsto m.mentries[1].term,$
$leaderId \mapsto leaderId,$
$ackCount \mapsto 0,$
$acksFrom \mapsto \{\}]\}$
$\land$ $entryCommitStats' =$
IF $entryKey \in$ DOMAIN $entryCommitStats \land \neg entryCommitStats[entryKey].committed$
THEN $[entryCommitStats$ EXCEPT $![entryKey].sentCount = @ + Cardinality(followers)]$
ELSE $entryCommitStats$
$\land$ UNCHANGED $\langle serverVars, candidateVars, leaderVars, logVars,$
$leaderCount, maxc, hovercraftVars,$
$netAggIndex, netAggMatchIndex, netAggCommitIndex,$
$netAggCurrentLeaderTerm, Servers\rangle$

NetAgg receives *AppendEntries* response from follower
$NetAggHandleAppendEntriesResponse(m) \triangleq$
$\land$ $m.mtype = AppendEntriesResponse$
$\land$ $m.mdest = netAggIndex$ Message is for *NetAgg*
$\land$ $netAggCurrentLeaderTerm \neq Nil$ *NetAgg* must be active
$\land$ $m.msuccess$ Process successful *ACKs*, *NACKs* go to *Leader* for point to point recovery
$\land$ $\exists\, pending \in netAggPendingEntries :$

```
∧ m.msource ∈ (Servers \ {pending.leaderId})
  Response is from a follower (in Servers) of the leader for this pending entry
∧ m.mmatchIndex ≥ pending.entryIndex
  Follower acknowledged this entry (or beyond)
∧ m.msource ∉ pending.acksFrom
  This is a new ACK from this follower for this item
∧ (IF m.msource ∉ DOMAIN netAggMatchIndex
     THEN PrintT("DEBUG: m.msource NOT IN DOMAIN netAggMatchIndex"
                 ∘ "\n  m.msource = " ∘ ToString(m.msource)
                 ∘ "\n  m = " ∘ ToString(m)
                 ∘ "\n  pending = " ∘ ToString(pending)
                 ∘ "\n  netAggMatchIndex = " ∘ ToString(netAggMatchIndex)
                 ∘ "\n  DOMAIN netAggMatchIndex = " ∘ ToString(DOMAIN netAggMatchIndex)
                 ∘ "\n  Servers = " ∘ ToString(Servers)
                 ∘ "\n  netAggCurrentLeaderTerm = " ∘ ToString(netAggCurrentLeaderTerm)
                 ∘ "\n  netAggPendingEntries = " ∘ ToString(netAggPendingEntries)
                 ∘ "\n  currentTerm = " ∘ ToString(currentTerm)
                 ∘ "\n  state = " ∘ ToString(state)
                 ∘ "\n  log length for m.msource = " ∘ ToString(Len(log[m.msource]))
                 ∘ "\n  commitIndex for m.msource = " ∘ ToString(commitIndex[m.msource])
                 ∘ "\n  All messages = " ∘ ToString(messages)  Might be very verbose
               )
     ELSE TRUE)
∧ LET updatedPending ≜ [pending EXCEPT !.acksFrom = @ ∪ {m.msource}]
      RequiredFollowerAcks ≜ Cardinality(Servers) ÷ 2
        Leader has one, need this many more from followers
  IN
   ∧ m.msource ∈ DOMAIN netAggMatchIndex
   ∧ netAggMatchIndex' = [netAggMatchIndex EXCEPT ![m.msource] = m.mmatchIndex]
   ∧ IF Cardinality(updatedPending.acksFrom) ≥ RequiredFollowerAcks
       THEN  Majority reached, send AGG_COMMIT to all Raft Servers
       LET   AggCommitMsgsSet ≜ {[mtype        ↦ AggCommit,
                                  mcommitIndex ↦ pending.entryIndex,
                                  msource      ↦ netAggIndex,
                                  mdest        ↦ srv,
                                  mterm        ↦ pending.entryTerm]
                                 : srv ∈ Servers}  Send to all Raft servers
              Messages that were valid, excluding the one we just processed
             RemainingActiveMessages ≜ ValidMessage(WithoutMessage(m, messages))
       IN
        ∧ messages' = [msgRec ∈ RemainingActiveMessages ∪ AggCommitMsgsSet ↦ 1]
This creates the new message bag:
- 'm' is effectively removed (as it's not in RemainingActiveMessages).
- All messages in AggCommitMsgsSet are added (or kept if already there by chance).
- All other previously active messages are preserved.
```

- All messages in the resulting bag have count 1, respecting *MyConstraint*.

$\land\ netAggPendingEntries' = netAggPendingEntries \setminus \{pending\}$

Remove committed entry from pending

$\land\ netAggCommitIndex' = Max(\{netAggCommitIndex,\ pending.entryIndex\})$

$\land$ UNCHANGED $\langle serverVars,\ candidateVars,\ leaderVars,\ logVars,\ instrumentationVars,\ hovercraftVars,\ netAggIndex,\ netAggCurrentLeaderTerm,\ Servers\rangle$

ELSE Majority not yet reached

$\land\ Discard(m)$ This defines $messages' = WithoutMessage(m,\ messages)$

$\land\ netAggPendingEntries' = (netAggPendingEntries \setminus \{pending\}) \cup \{updatedPending\}$

$\land$ UNCHANGED $netAggCommitIndex$

$\land$ UNCHANGED $\langle serverVars,\ candidateVars,\ leaderVars,\ logVars,\ instrumentationVars,\ hovercraftVars,\ netAggIndex,\ Servers,\ netAggCurrentLeaderTerm\rangle$

Server receives *AGG_COMMIT* from *NetAgg*

$HandleAggCommit(i,\ m) \triangleq$

$\land\ m.mtype = AggCommit$

$\land\ m.mdest = i$

$\land\ state[i] \in \{Leader,\ Follower\}$

$\land\ ((state[i] = Leader \quad \land\ m.mterm = currentTerm[i]) \lor$

Leader: entry's term must match leader's current term

$(state[i] = Follower \quad \land\ m.mterm \leq currentTerm[i]))$

Follower: entry's term can be current or older (but not newer)

$\land$ LET $receivedCommitIndex \triangleq m.mcommitIndex$ Added for clarity

$currentLogLen \triangleq Len(log[i])$ Added: get current *log* length

$newAdvancedCommitIndex \triangleq Max(\{commitIndex[i],\ receivedCommitIndex\})$

Renamed & Logic: advance if *m.mcommitIndex* is higher

$newCommitIndex \triangleq Min(\{newAdvancedCommitIndex,\ currentLogLen\})$

Modified: cap at current *log* length

$committedIndexes \triangleq \{k \in Nat : \ \land\ k > commitIndex[i] \ \land\ k \leq newCommitIndex\}$

$keysToUpdate \triangleq$ IF $state[i] = Leader$
THEN $\{key \in$ DOMAIN $entryCommitStats : key[1] \in committedIndexes\}$
ELSE $\{\}$

IN

$\land\ commitIndex' = [commitIndex$ EXCEPT $![i] = newCommitIndex]$

$\land\ entryCommitStats' =$ IF $state[i] = Leader$
THEN $[key \in$ DOMAIN $entryCommitStats \mapsto$
IF $key \in keysToUpdate$
THEN $[entryCommitStats[key]$ EXCEPT $!.committed =$ TRUE$]$
ELSE $entryCommitStats[key]]$
ELSE $entryCommitStats$

$$
\begin{array}{l}
\quad \land \text{IF } state[i] = Leader \\
\qquad \text{THEN } \land nextIndex' = [nextIndex \text{ EXCEPT } ![i] = \\
\qquad\qquad\qquad [j \in Server \mapsto \\
\qquad\qquad\qquad\quad \text{IF } j \in Servers \setminus \{i\} \\
\qquad\qquad\qquad\quad\quad \text{THEN } Max(\{nextIndex[i][j], newCommitIndex + 1\}) \\
\qquad\qquad\qquad\quad\quad \text{ELSE } nextIndex[i][j]]] \\
\qquad\qquad \land matchIndex' = [matchIndex \text{ EXCEPT } ![i] = \\
\qquad\qquad\qquad [j \in Server \mapsto \\
\qquad\qquad\qquad\quad \text{IF } j \in Servers \setminus \{i\} \\
\qquad\qquad\qquad\quad\quad \text{THEN } Max(\{matchIndex[i][j], newCommitIndex\}) \\
\qquad\qquad\qquad\quad\quad \text{ELSE } matchIndex[i][j]]] \\
\qquad \text{ELSE } \text{UNCHANGED } \langle nextIndex, matchIndex \rangle \\
\land Discard(m) \\
\land \text{UNCHANGED } \langle serverVars, candidateVars, log, maxc, leaderCount, \\
\qquad\qquad hovercraftVars, netAggVars, Servers, netAggVars \rangle
\end{array}
$$

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.
fails when *Leader* restarts

$$
\begin{array}{l}
HandleAppendEntriesRequest(i, j, m) \triangleq \\
\quad \text{LET } logOk \triangleq \lor m.mprevLogIndex = 0 \\
\qquad\qquad\qquad \lor \land m.mprevLogIndex > 0 \\
\qquad\qquad\qquad\quad \land m.mprevLogIndex \leq Len(log[i]) \\
\qquad\qquad\qquad\quad \land m.mprevLogTerm = log[i][m.mprevLogIndex].term \\
\qquad rejectHovercraftMismatchCondition \triangleq \\
\qquad\quad \land m.mentries \neq \langle\rangle \\
\qquad\quad \land \text{LET } entry \triangleq m.mentries[1] \\
\qquad\qquad\quad v \triangleq entry.value \\
\qquad\qquad\quad msgTerm \triangleq entry.term \\
\qquad\qquad \text{IN } \neg(\land \langle v, msgTerm \rangle \in unorderedRequests[i] \\
\qquad\qquad\qquad \land \langle v, msgTerm \rangle \in \text{DOMAIN } switchBuffer \\
\qquad\qquad\qquad \land switchBuffer[\langle v, msgTerm \rangle].term = msgTerm)
\end{array}
$$

Condition that triggers the CHOOSE for the leader; corner case *NACK*

$$isReplyToLeaderCase \triangleq rejectHovercraftMismatchCondition \lor \neg logOk$$

Check if a leader exists to be chosen for the reply

$$canChooseLeaderForReply \triangleq \exists\, l\_exists \in Servers : state[l\_exists] = Leader$$

$$
\begin{array}{l}
\quad \text{IN } \land m.mterm \leq currentTerm[i] \\
\qquad\quad \land \lor \land \text{ reject request branch} \\
\qquad\qquad\qquad \land (\text{ conditions for rejecting the request} \\
\qquad\qquad\qquad\quad \lor m.mterm < currentTerm[i] \\
\qquad\qquad\qquad\quad \lor \land m.mterm = currentTerm[i]
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad \land\ state[i] = Follower \\
\qquad\qquad \land\ \neg logOk \\
\qquad \lor\ \land\ m.mterm = currentTerm[i] \\
\qquad\qquad \land\ state[i] = Follower \\
\qquad\qquad \land\ rejectHovercraftMismatchCondition \\
\quad)
\end{array}
$$

$$
\begin{array}{l}
\land\ \text{LET } respondTo \triangleq \text{IF } isReplyToLeaderCase \\
\qquad\qquad\qquad\qquad \text{THEN CHOOSE } l \in Servers : state[l] = Leader \\
\qquad\qquad\qquad\qquad \text{ELSE } m.msource \\
\quad \text{IN } Reply([mtype \mapsto AppendEntriesResponse, \\
\qquad\qquad\quad mterm \mapsto currentTerm[i], \\
\qquad\qquad\quad msuccess \mapsto \text{FALSE}, \\
\qquad\qquad\quad mmatchIndex \mapsto 0, \\
\qquad\qquad\quad msource \mapsto i, \\
\qquad\qquad\quad mdest \mapsto respondTo], \\
\qquad\qquad\quad m) \\
\land\ \text{UNCHANGED } \langle serverVars, logVars, unorderedRequests \rangle
\end{array}
$$

$\lor$ return to follower state

$$
\begin{array}{l}
\land\ m.mterm = currentTerm[i] \\
\land\ state[i] = Candidate \\
\land\ state' = [state \text{ EXCEPT } ![i] = Follower] \\
\land\ \text{UNCHANGED } \langle currentTerm, votedFor, logVars, messages, \\
\qquad\qquad\qquad unorderedRequests \rangle
\end{array}
$$

$\lor$ accept request

$$
\begin{array}{l}
\land\ m.mterm = currentTerm[i] \\
\land\ state[i] = Follower \\
\land\ logOk \\
\land\ \text{LET } index \triangleq m.mprevLogIndex + 1 \\
\qquad\quad respondToIfAccepted \triangleq m.msource \quad \text{respondTo will be m.msource here} \\
\quad \text{IN } \lor \quad \text{already done with request or empty entries} \\
\qquad\qquad \land\ \lor\ m.mentries = \langle\rangle \\
\qquad\qquad\quad \lor\ \land\ m.mentries \neq \langle\rangle \\
\qquad\qquad\qquad \land\ Len(log[i]) \geq index \\
\qquad\qquad\qquad \land\ log[i][index].term = m.mentries[1].term \\
\qquad\qquad \land\ commitIndex' = [commitIndex \text{ EXCEPT } ![i] = m.mcommitIndex] \\
\qquad\qquad \land\ Reply([mtype \mapsto AppendEntriesResponse, \\
\qquad\qquad\qquad\quad mterm \mapsto currentTerm[i], \\
\qquad\qquad\qquad\quad msuccess \mapsto \text{TRUE}, \\
\qquad\qquad\qquad\quad mmatchIndex \mapsto m.mprevLogIndex + Len(m.mentries), \\
\qquad\qquad\qquad\quad msource \mapsto i, \\
\qquad\qquad\qquad\quad mdest \mapsto respondToIfAccepted], \\
\qquad\qquad\qquad\quad m) \\
\qquad\qquad \land\ \text{UNCHANGED } \langle serverVars, log, unorderedRequests \rangle
\end{array}
$$

```
                  ∨ conflict: remove 1 entry
                    ∧ m.mentries ≠ ⟨⟩
                    ∧ Len(log[i]) ≥ index
                    ∧ log[i][index].term ≠ m.mentries[1].term
                    ∧ LET newLog ≜ SubSeq(log[i], 1, index − 1)
                      IN  log′ = [log EXCEPT ![i] = newLog]
                    ∧ UNCHANGED ⟨serverVars, commitIndex, messages, unorderedRequests⟩
                  ∨ no conflict: append entry
                    ∧ m.mentries ≠ ⟨⟩
                    ∧ Len(log[i]) = m.mprevLogIndex
                    ∧ ¬rejectHovercraftMismatchCondition
                    ∧ LET entryMetadata ≜ m.mentries[1]
                          vt ≜ ⟨entryMetadata.value, entryMetadata.term⟩
                          fullEntryFromCache ≜ switchBuffer[vt]
                          entryForLocalLog ≜ [term  ↦ entryMetadata.term,
                                              value ↦ entryMetadata.value,
                                              payload ↦ fullEntryFromCache.payload]
                      IN  log′ = [log EXCEPT ![i] = Append(log[i], entryForLocalLog)]
                          ∧ unorderedRequests′ = [unorderedRequests EXCEPT ![i] = @ \ {vt}]
                    ∧ UNCHANGED ⟨serverVars, commitIndex, messages⟩
         ∧ UNCHANGED ⟨candidateVars, leaderVars, instrumentationVars,
            switchBuffer, switchIndex, switchSentRecord, Servers, netAggVars⟩

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
        ∧ LET
              newMatchIndex ≜ m.mmatchIndex
              entryKey ≜ IF newMatchIndex > 0 ∧ newMatchIndex ≤ Len(log[i])
                              THEN ⟨newMatchIndex, log[i][newMatchIndex].term⟩
                              ELSE ⟨0, 0⟩  Invalid index or empty log
          IN  ∧ nextIndex′  = [nextIndex  EXCEPT ![i][j] = m.mmatchIndex + 1]
              ∧ matchIndex′ = [matchIndex EXCEPT ![i][j] = m.mmatchIndex]
              ∧ entryCommitStats′ =
                  IF ∧ entryKey ≠ ⟨0, 0⟩
                     ∧ entryKey ∈ DOMAIN entryCommitStats
                     ∧ ¬entryCommitStats[entryKey].committed
                  THEN [entryCommitStats EXCEPT ![entryKey].ackCount = @ + 1]
                  ELSE entryCommitStats
      ∨ ∧ ¬m.msuccess  not successful
        ∧ nextIndex′ = [nextIndex EXCEPT ![i][j] =
                          Max({nextIndex[i][j] − 1, 1})]
        ∧ UNCHANGED ⟨matchIndex, entryCommitStats⟩
```

$\land\ Discard(m)$
$\land$ UNCHANGED $\langle serverVars,\ candidateVars,\ logVars,\ maxc,\ leaderCount,$
$\quad hovercraftVars,\ Servers,\ netAggVars \rangle$

Network state transitions

The network duplicates a message
$DuplicateMessage(m) \triangleq$
$\quad \land\ Send(m)$
$\quad \land$ UNCHANGED $\langle serverVars,\ candidateVars,\ leaderVars,\ logVars,$
$\qquad instrumentationVars,\ hovercraftVars,\ Servers,\ netAggVars \rangle$

The network drops a message
$DropMessage(m) \triangleq$
$\quad \land\ Discard(m)$
$\quad \land$ UNCHANGED $\langle serverVars,\ candidateVars,\ leaderVars,\ logVars,$
$\qquad instrumentationVars,\ hovercraftVars,\ Servers,\ netAggVars \rangle$

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

Modified. Leader i sends j an *AppendEntries* request containing exactly 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. Sending empty entries is done for telling followers *Leader* is alive.
$AppendEntries(i,\ j) \triangleq$
$\quad \land\ i \neq j$
$\quad \land\ state[i] = Leader$
$\quad \land\ Len(log[i]) > 0$
$\quad$ Only proceed if the leader has entries to send
$\quad \land\ nextIndex[i][j] \leq Len(log[i])$

```
    Only proceed if there are entries to send to this follower
    ∧ matchIndex[i][j] < nextIndex[i][j]
    Only send if follower hasn't already acknowledged this index
    ∧ LET entryIndex ≜ nextIndex[i][j]
          entry ≜ log[i][entryIndex]
          entryMetadata ≜ [term ↦ entry.term, value ↦ entry.value]
          entries ≜ ⟨entryMetadata⟩
          entryKey ≜ ⟨entryIndex, entry.term⟩
          prevLogIndex ≜ entryIndex − 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]})
           entries ≜ SubSeq(log[i], nextIndex[i][j], lastEntry)

      IN  Send([mtype          ↦ AppendEntriesRequest,
                mterm          ↦ currentTerm[i],
                mprevLogIndex  ↦ prevLogIndex,
                mprevLogTerm   ↦ prevLogTerm,
                mentries       ↦ entries,
                 mlog is used as a history variable for the proof.
                 It would not exist in a real implementation.
                mlog           ↦ log[i],
                mcommitIndex   ↦ Min({commitIndex[i], entryIndex}),
                msource        ↦ i,
                mdest          ↦ j])
      ∧   entryCommitStats' =
            IF entryKey ∈ DOMAIN entryCommitStats
                ∧ ¬entryCommitStats[entryKey].committed
             THEN [entryCommitStats EXCEPT ![entryKey].sentCount = @ + 1]
             ELSE entryCommitStats
    ∧ UNCHANGED ⟨serverVars, candidateVars, leaderVars, logVars, maxc,
                 leaderCount, hovercraftVars, Servers, netAggVars⟩

MySwitchPlusPlusNext ≜
   Switch actions (client request handling)
   ∨ ∃ i ∈ Servers, v ∈ Value :
       state[i] = Leader ∧ SwitchClientRequest(switchIndex, i, v)

   ∨ ∃ v ∈ DOMAIN switchBuffer :
       SwitchClientRequestReplicateAll(switchIndex, v)

   ∨ ∃ i ∈ Servers, v ∈ DOMAIN switchBuffer :
       state[i] = Leader ∧ LeaderIngestHovercRaftRequest(i, v)
```

NetAgg path: *Leader* sends to *NetAgg* instead of direct *AppendEntries*

$\lor \exists\, i \in Servers :$
$\quad state[i] = Leader \land AppendEntriesToNetAgg(i)$

$\lor \exists\, m \in \{msg \in ValidMessage(messages) :$
$\quad msg.mtype = AppendEntriesNetAggRequest\} : NetAggForwardAppendEntriesAll(m)$

Regular message handling (for *AppendEntries* from *NetAgg* to followers)
Our spec assumes an aggregator *NetAgg* is available

$\lor \exists\, m \in \{msg \in ValidMessage(messages) :$
$\quad msg.mtype \in \{AppendEntriesRequest\}\} :$
$\quad Receive(m)$

$\lor \exists\, m \in \{msg \in ValidMessage(messages) :$
$\quad msg.mtype = AppendEntriesResponse \land$
$\quad msg.mdest = netAggIndex\} :$
$\quad NetAggHandleAppendEntriesResponse(m)$

Handle *AGG_COMMIT*

$\lor \exists\, i \in Servers,\ m \in \{msg \in ValidMessage(messages) :$
$\quad msg.mtype = AggCommit\} : m.mdest = i \land HandleAggCommit(i, m)$

Handle *AppendEntriesResponse* failing messages that go to leader
to be enabled for point to point recovery *todo*!

$\lor \exists\, m \in \{msg \in ValidMessage(messages) :$
$\quad msg.mtype = AppendEntriesResponse \land$
$\quad msg.mdest \in Servers \land state[msg.mdest] = Leader\} :$
$\quad \text{LET } i \triangleq m.mdest$
$\quad\quad\quad j \triangleq m.msource$
$\quad \text{IN } \; AppendEntries(i, j)$

Leader doesn't use *AdvanceCommitIndex* in *HovercRaft* ++
Commit advancement happens via *AGG_COMMIT*

Server crash and recovery actions for the bonus exercise

$\lor \exists\, i \in Servers : Restart(i)$ Allow servers to crash and restart

Leader election actions (optional for handling leader crashes)

$\lor \exists\, i \in Servers : Timeout(i)$ Allow followers to timeout and start elections
$\lor \exists\, i \in Servers : BecomeLeader(i)$ Allow candidates to become leaders
$\lor \exists\, i, j \in Servers : RequestVote(i, j)$ Allow vote requests
$\lor \exists\, m \in \{msg \in ValidMessage(messages) :$
$\quad msg.mtype \in \{RequestVoteRequest, RequestVoteResponse\}\} :$
$\quad Receive(m)$ Handle vote messages

Follower request dropping (for testing failure scenarios)

$\lor \exists\, i \in Servers : \exists\, vt \in unorderedRequests[i] :$
$\quad state[i] = Follower \land FollowerDropRequest(i, vt)$

$MySwitchPlusPlusSpec \triangleq MyInit \land \Box[MySwitchPlusPlusNext]_{vars}$
$Spec \triangleq Init \land \Box[MySwitchPlusPlusNext]_{vars}$

—————————— Invariants ——————————

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

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 preceding entries."

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

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)

$$
\begin{aligned}
&LeaderCompletenessInv \triangleq \\
&\quad \forall i \in Server : \\
&\qquad state[i] = Leader \Rightarrow \\
&\qquad \forall j \in Server : i \neq j \Rightarrow \\
&\qquad\quad CheckIsPrefix(CommittedTermPrefix(j, currentTerm[i]), log[i])
\end{aligned}
$$

Committed *log* entries should never conflict between servers

$$
\begin{aligned}
&LogInv \triangleq \\
&\quad \forall i, j \in Server : \\
&\qquad \lor CheckIsPrefix(Committed(i), Committed(j)) \\
&\qquad \lor CheckIsPrefix(Committed(j), Committed(i))
\end{aligned}
$$

Note that *LogInv* checks for safety violations across space
This is a key safety invariant and should always be checked

$$
\begin{aligned}
&\textsc{theorem}\ MySwitchPlusPlusSpec \Rightarrow (\Box LogInv \land \Box LeaderCompletenessInv \\
&\qquad\qquad\qquad \land \Box LogMatchingInv \land \Box MoreThanOneLeaderInv)
\end{aligned}
$$

fake inv to obtain a trace and observe progress for client requests advancing to committed.

$$
\begin{aligned}
&LeaderCommitted \triangleq \\
&\quad \exists i \in Servers : commitIndex[i] \neq 2
\end{aligned}
$$

References

[1] Kogias, M., Bugnion, E.: Hovercraft: achieving scalability and fault-tolerance for microsecond-scale datacenter services. In: Proceedings of the Fifteenth European Conference on Computer Systems. EuroSys '20. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3342195.3387545 . https://doi.org/10.1145/3342195.3387545

[2] Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference. USENIX ATC'14, pp. 305–320. USENIX Association, USA (2014)

[3] Kogias, M., Prekas, G., Ghosn, A., Fietz, J., Bugnion, E.: R2p2: making rpcs first-class datacenter citizens. In: Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference. USENIX ATC '19, pp. 863–879. USENIX Association, USA (2019)