Accelerating Reads with Consistency-Aware Network Routing

Technical Report

This technical report includes the following additional material compared to our SOSP #11 submission: a detailed proof of FLAIR safety, additional evaluation results with five-replica deployment, and the TLA+ specification.

1 FLAIR Correctness

FLAIR uses the underlying leader-based consensus protocol to process write requests and to serve reads from the leader. FLAIR extends these protocols with the ability to serve reads from followers. The rest of this proof focuses on proving the safety of reads in FLAIR.

Consensus protocol properties. First, we state the main properties of a target leader-based consensus protocol:

Property 1. At any time, there is at most one active leader in the system, that is followed by the majority of the nodes, and is the only node that can commit new values. This leader has the highest term id in the system.

Property 2. Reads processed by the leader are always linearizable.

Property 3. If an operation at index i in the log is committed, then every operation with an index smaller than i is committed as well.

Property 4. If a follower accepts a new entry to its log, then it is guaranteed that the follower log is identical to the leader's log up to that entry.

We note that all major leader-based consensus protocols (e.g., Raft [1], Viewstamped replication [2, 3], DARE [4], Zookeeper [5], and multi-Paxos implementations [6, 7]) hold these properties.

Definitions. Before proving that FLAIR guarantees safety, we need to define a few properties.

Definition. We say the switch is *active* if it has an active leader-switch session. If the switch is not active, it will drop all FLAIR requests and replies, rendering the system unavailable.

Consequently, our proof focuses on proving safety when the switch is active.

Definition. We say a kgroup is *stable* when there are no outstanding write requests in the system that *may modify* the objects in the kgroup.

Definition. We say a request or a reply is *valid* when the term id, and session id in the request match the leader information and in the reply match the switch information. Invalid requests/replies are dropped. Replicas use the information in the request to fill the fields of the reply. Consequently, request-reply pairs that span multiple sessions are dropped.

Our proof focuses on proving safety with valid requests/replies.

Assumptions. FLAIR assumes the following environment properties (Note that the underlying consensus protocol may have more stringent assumptions):

- The network is unreliable and asynchronous, as there are no guarantees that packets will be received in a timely manner or delivered at all.
- There is no bound on the time a node or the switch takes to process a packet.
- Clocks are not synchronized, and there is no bound on the clock's drift rate.
- Nodes fail following the fail-stop model in which nodes may stop working but will never send erroneous messages (i.e., no byzantine failures).

Moreover, we need to define a few terms. In the following all times are relative to the switch's clock:

- $time(w)$ is the time a request/reply w was processed by the switch.
- $seq(w)$ is the sequence number of a request w.
- $wlseq_{switch}(t)$ is the largest sequence number issued by the switch of all write requests that have been received and processed by the switch before or at time t, i.e., $wlseq_{switch}(t)$ is the sequence number of a write request $w_j \mid seq(w_j) > seq(w_i)\ \forall\ w_i \neq w_j$ and $time(w_i) \leq t$.
- $rlseq_{switch}(t)$ is the largest sequence number of all write replies that have been processed by the switch before or at time t, i.e., $rlseq_{switch}(t)$ is the sequence number of a write reply $r_j \mid seq(r_j) > seq(r_i)\ \forall\ r_i \neq r_j$ and $time(r_i) \leq t$.
- $lseq_{leader}(t)$ is the largest sequence number of all write requests that have been processed by the leader before or at time t (time relative to the switch's clock).

Now, we have all the definitions to present our proof.

We will prove the safety property for the simple case in which there is a single kgroup in the system, and the kgroup has a single object (*obj*). All read and write requests access this single object. At the end of the proof, we will generalize it to multiple kgroups with multiple objects.

When the switch is active, the kgroup can be in one of two states: unstable or stable. Requests to unstable kgroups are forwarded to the leader and therefore are linearizable (Property 2). For stable kgroup, the switch forwards the read request to one of the followers included in the consistent_followers bitmap. To prove FLAIR safety for the stable kgroup we need to prove that while a kgroup is stable, the value of the kgroup object does not change and the followers included in the consistent_followers bitmap in the kgroup entry hold the latest committed value of the object (Section 1.2), then we need to prove that read requests processed by the followers are safe (Section 1.3).

1.1 Session Start Process

The safety of FLAIR relies on the following theorem.

Theorem 1. At any moment in time there is at most one switch that is accessible by the FLAIR packets and has an active session.

Proof. Proof by contradiction. Let's assume at time t_o is the first moment in which the system had two active and accessible switches s_a, and s_b. Without loss of generality, let's assume s_b is the later switch to be activated by the leader. This means the leader has just started a new session with s_b. But before starting a session the leader asks the central controller to neutralize the old session, meaning reroute all FLAIR traffic from the old switch to the new one. Consequently, it is not possible for s_a to be accessible by FLAIR packets. □

1.2 Kgroup Stability

Lemma 1. At any moment in time t_o the following inequality holds: $wlseq_{switch}(t_o) \geq lseq_{leader}(t_o) \geq rlseq_{switch}(t_o)$

Proof. The switch sequentially processes all write requests and assigns a unique and strictly increasing sequence number for every write request. For the left side of the inequality, the switch always processes a write request before sending it to the leader. Hence, at all times, $wlseq_{switch}(t_o) \geq lseq_{leader}$. For the right side of the inequality, the leader will receive the write request and processes it before sending a reply. A write reply has the same sequence number as the corresponding write request. Hence, at all times, $lseq_{leader}(t_o) \geq rlseq_{switch}(t_o)$. □

Lemma 1 implies the following corollary:

Corollary 1. If at time t_o $wlseq_{switch}(t_o) = rlseq_{switch}(t_o)$, then $wlseq_{switch}(t_o) = lseq_{leader}(t_o) = rlseq_{switch}(t_o)$. □

Lemma 2. At any moment in time t_o, if a request w_l has a sequence number $seq(w_l) = wlseq_{switch}(t_o)$, then w_l is the last write request processed by the switch up to time t_o, and sequence number in the kgroup entry seq_num = $seq(w_l)$.

Proof. The switch processes all packets sequentially in a pipeline. On every write, the switch atomically increments the session_seq_num in the session array, marks the kgroup entry unstable, and updates the seq_num in the kgroup entry. The fact that request w_l has the largest sequence number signifies that it was the last request to be processed by the switch up to time t_o. □

Lemma 3. If at time t_o $wlseq_{switch}(t_o) = rlseq_{switch}(t_o)$, then the last committed value of *obj* at time t_o is the value written by the request w_l with $seq(w_l) = wlseq_{switch}(t_o)$.

Proof. The fact that $wlseq_{switch}(t_o) = rlseq_{switch}(t_o)$ implies that $seq(w_l) = wlseq_{switch}(t_o) = lseq_{leader}(t_o)$ (Corollary 1), meaning that w_l is the last write request that has been processed by the leader up to time t_o.

At all times, the leader keeps track of the largest sequence number (largest_seq_num = $lseq_{leader}(t)$) processed in the current session. The leader drops every write request with a sequence number smaller than largest_seq_num. Consequently, regardless of the order in which the write requests are processed, when the leader processes w_l, it will set the largest_seq_num to equal $seq(w_l)$ and will drop any unprocessed requests with $time(w) < t_o$. Consequently, at time t_o, the last committed value is the value written by w_l. □

Now we can prove our main object stability lemma.

Lemma 4. In any time interval $[t_o, t_1]$, if $wlseq_{switch}(t_o) = rlseq_{switch}(t_o) = wlseq_{switch}(t_1)$, then the object *obj* is stable (was not modified) in the period $[t_o, t_1]$.

Proof. Assume that the write request w_l has a $seq(w_l) = wlseq_{switch}(t_1)$. The fact that $wlseq_{switch}(t_o) = rlseq_{switch}(t_o)$ implies that w_l is last request that has been processed by the leader up to time t_o (Lemma 3).

The fact that $seq(w_l) = wlseq_{switch}(t_o) = wlseq_{switch}(t_1)$ signifies that no new write requests have been processed by the switch in the interval $[t_o, t_1]$, and w_l is still the last request that has been processed by the leader up to time t_1. Consequently, the value of the object did not change in the interval $[t_o, t_1]$. □

Now we prove the stability of the kgroup data.

Lemma 5. If at time t_o $wlseq_{switch}(t_o) = rlseq_{switch}(t_o)$, then the kgroup is stable and the fields of the kgroup entry (consistent_followers bitmap and log_indx) have values equal to the fields of the write reply r_l with $seq(r_l) = wlseq_{switch}(t_o)$.

Proof. Assume that a write request w_l has $seq(w_l) = wlseq_{switch}(t_o)$ with a corresponding reply r_l with $seq(r_l) = seq(w_l)$, then seq_num in the kgroup entry at time t_o equals $seq(w_l)$ (Lemma 2).

The only time the switch will mark a kgroup as stable and update the consistent_followers bitmap and log_indx fields is when the switch receives a write reply with a sequence number equal to the seq_num in the kgroup entry.

For all other write replies with $seq(r) \neq$ seq_num, the switch will forward them to the client without updating the kgroup entry. Consequently, at time t_o, the last values of the consistent_followers and log_indx fields in the kgroup entry are the value written by r_l. □

Lemma 6. In any time interval $[t_o, t_1]$, if $wlseq_{switch}(t_o) = rlseq_{switch}(t_o) = wlseq_{switch}(t_1)$, then the kgroup is marked stable in the period $[t_o, t_1]$ and the kgroup fields (consistent_followers bitmap and log_indx) did not change in this period.

Proof. Assume that a write reply r_l has a sequence number $seq(r_l) = wlseq_{switch}(t_o) = rlseq_{switch}(t_o)$. This signifies that when the switch processed r_l at $time(r_l) \leq t_o$, it marked the kgroup stable and set the kgroup entry fields to the values in the r_l fields (Lemma 5). The fact that $seq(r_l) = wlseq_{switch}(t_1)$ signifies that r_l is still the last reply processed by the switch at time t_1 and the kgroup is still stable. □

Now we have all the facts to prove the main stability property.

Theorem 2 (Object Stability). During any period $[t_o, t_1]$ in which a kgroup is marked stable, there are no updates to the kgroup object, and the followers included in the consistent_followers field have the latest committed value for *obj*.

Proof. In any time interval $[t_o, t_1]$, the kgroup is stable iff $wlseq_{switch}(t_o) = rlseq_{switch}(t_o) = wlseq_{switch}(t_1)$ (Lemma 6), the object *obj* is stable in the period $[t_o, t_1]$ (Lemma 4), and the value of the consistent_followers bitmap is stable, and has the value of a write r_l with $seq(r_l) = wlseq_{switch}(t_1)$ (Lemma 6).

A leader will include a follower in the consistent_followers only if the follower acknowledges the write operation. Following from Properties 3 and 4, those followers that acknowledged the write have an identical log to the leader's up to that log entry and hence have a consistent value for the object. □

1.3 Safety

The switch forwards a read request for a stable kgroup to one of the consistent followers. The reply from those followers is linearizable unless the object has been modified after the read request is processed by the switch and before the switch receives the follower's reply. To preserve safety, the switch performs a safety check on every read reply to detect stale replies.

Theorem 3 (Follower Read Reply Safety). The follower's read replies that the switch forwards to the client are linearizable.

Proof. The leader will send a read request to one of the followers in the consistent_followers bitmap only if a kgroup is stable. Those followers have a consistent version of the object that is identical to the leader's version (Theorem 1).

The switch sets the sequence number of a read request to match the sequence number of the kgroup entry. Also, the follower sets the SEQ sequence number of read replies to equal the SEQ sequence number of the corresponding read request.

The switch will only forward a read reply to a client from a follower if the reply passes the following safety check: The kgroup is stable and the sequence number of the reply matches the sequence number of the kgroup. This indicates that no writes occurred since the read was processed by the switch and the object is still consistent at the follower. □

Now we have all the facts to prove the main safety theorem.

Theorem 4 (Read Safety). FLAIR guarantees linearizability of client reads at all times.

Proof. The switch will only process requests when it is active. When the switch is active, a kgroup can be in one of two states: unstable or stable. When a kgroup is not stable, reads are linearizable as they are processed by the leader (Property 2). When a kgroup is stable, the switch will forward requests to one of the followers included in the consistent_followers field. When the switch receives the read reply, if the reply passes the safety check, it is forwarded to the client and is linearizable (Theorem 2). If a read reply does not pass the safety check, the switch will drop the reply and resubmit the read request to the leader. Leader read replies are always linearizable (Property 1). Consequently, FLAIR reads are always linearizable. □

Generalization.

Multi-kgroup support. FLAIR does not support multi-object transactions. FLAIR guarantees linearizability only per object and does not guarantee linearizability of operations spanning multiple objects.

Generalization to multiple objects per kgroup. The switch treats all the objects in a kgroup as a single object. If the switch receives a write operation to any object in a kgroup, the kgroup entry is marked unstable. The kgroup is marked stable only when the last write to the kgroup is acknowledged by the leader. Consequently, having multiple objects per kgroup can only affect performance as it can lead to marking an object unstable and forwarding its reads to the leader only because another object in the kgroup is being updated. These false positives do not affect safety □

2 Additional Evaluation

This sections presents additional evaluation results. In the following experiments we use the same setup, run the same workloads, and compare the same alternatives specified in our paper. The only difference here is that we use 5 nodes as servers and 8 servers to generate client workload.

In summary, the 5-replica results corroborate our findings in the paper.

2.1 Throughput Evaluation

Figure 1 shows the throughput of the six systems for workloads A, B, and C. For workload C (Figure 1.C), FlairKV and Leases achieve the highest throughput, 4.4 M op/s, as both systems can utilize all replicas to serve read requests. FlairKV and Leases achieve 4.7 times higher throughput relative to OptRaft, which only uses the leader to serve read requests. Finally, FlairKV and Leases achieve at

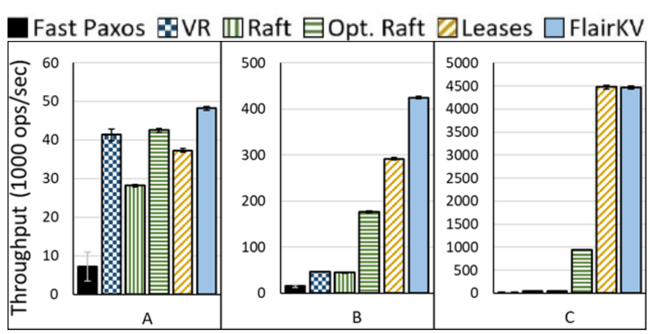
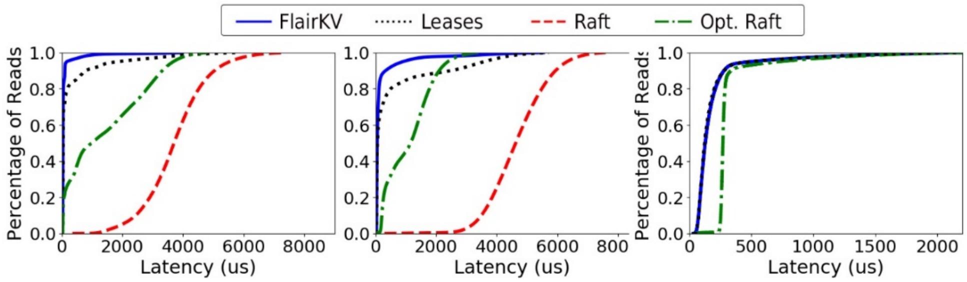
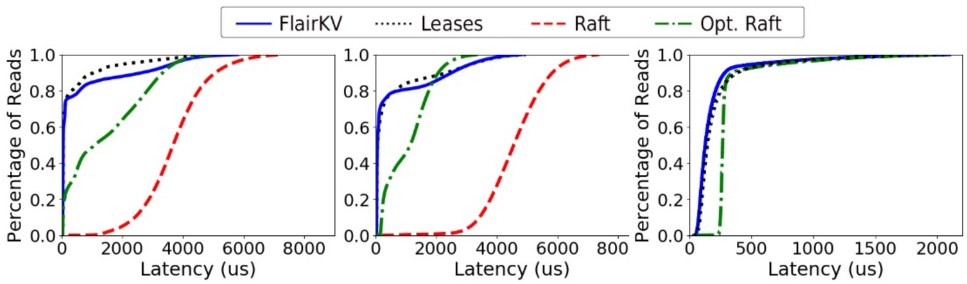


Figure 1. System's throughput using the YCSB workloads A, B, and C with uniform key popularity distribution. Error bars show standard deviation, which is less than 1% for all systems except Fast Paxos, which had higher variance.



(a) A-Uniform (b) B-Uniform (c) C-Uniform

Figure 3. Latency CDF. The figures shows the latency CDF for read requests using the Uniform distribution under workload A (a), B (b), and C (c).



(a) A-Zipf (b) B- Zipf (c) C- Zipf

Figure 4. Latency CDF. The figures shows the latency CDF for read requests using the Zipf distribution under workload A (a) and B (b), and C (c).

least 82 times higher throughput relative to Raft, VR, and Fast Paxos, as these protocols contact the majority of followers for every read operation.

Figure 1.B shows the throughput under workload B, with FlairKV achieving the highest performance. FlairKV achieves 46% higher throughput than Leases for three primary reasons. First, FlairKV uses the leader-avoidance load-balancing technique, which accelerates writes and reduces the time in which kgroups are marked unstable. We recorded the number of read requests served by the leader and found that it only served 2.9% of the read requests in FlairKV (those are reads to unstable kgroups), while it served 20% of the reads in Leases.

Second, when an object is not stable, Leases incurs extra latency, while FlairKV knows that the object is not stable and forwards the read requests for that object directly to the leader; Leases clients send the request to one of the followers, which redirects it to the leader. Third, Leases write operations need to reach all the followers, while

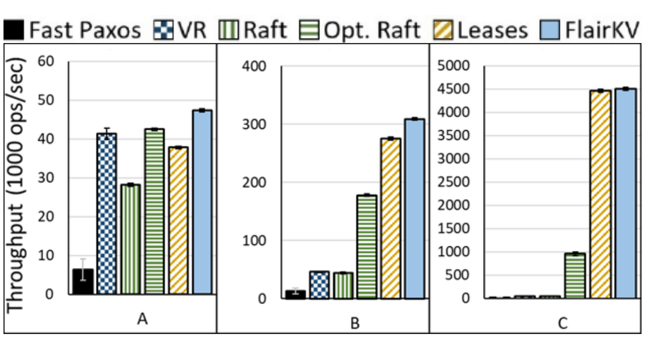


Figure 2. System's throughput using the YCSB workloads A, B, and C with Zipf key popularity distribution. Error bars show standard deviation which is less than 1% for all systems except Fast Paxos, which had higher variance.

FlairKV writes only need a majority. FlairKV achieves 2.4 times higher throughput than OptRaft, and at least 9 times higher than Raft, VR, and Fast Paxos.

Figure 1.A shows the throughput under write-intensive workload A. FlairKV achieves the highest performance, which is around 13% higher than Leases and OptRaft, and around 16% and 71% higher performance than VR and Raft, respectively. Fast Paxos achieves the lowest throughput. We note that the performances of Raft, VR, and Fast Paxos do not change significantly across the workloads, as reads still involve a majority of the followers.

Under the Zipf popularity distribution (Figure 2), FlairKV achieves comparable performance improvement, with a slight reduction in throughput under workloads A and B due to increased contention on the popular keys. Due to higher contention, 6.5% of the reads are redirected to the leader in Leases, compared to only 0.3% in FlairKV, as FlairkKV can detect that there is a concurrent write to an object and directly forward the read to the leader.

2.2 Latency Evaluation

We measured the operations latency under YCSB workloads A, B, and C. Figure 3 shows the latency of FlairKV, Leases, OptRaft, and Raft. Under the uniform distribution workload Figure 3 (a) and (b), FlairKV lowers the latency for the slowest 40% of operations by up to 81% relative to Leases. Figure 3 (c) shows that for a read-only workload, both FlairKV and Leases achieve similar latency, up to 52% lower than OptRaft. We excluded Raft from the figure because it had a poor performance under heavy read-only workload. Under the Zipf workload, FlairKV achieves up to 50% lower latency relative to Leases for workload B (Figure 4(b)). FlairKV and Leases achieve comparable latency under the write-heavy workload A. For workload C (Figure 4(c)), FlairKV achieved up to 28% lower latency than Leases for the slowest 20% of operations. Both FlairKV and Leases achieve more than 45% lower median latency than OptRaft.

Under all workloads, FlairKV significantly improves operation's latency relative to OptRaft and Raft. The median latency of FlairKV is 1.4% of Raft's latency and 4-6% of OptRaft's latency.

References

[1] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX Annual Technical Conference (USENIX ATC)*, 2014.

[2] B. Liskov and J. Cowling, "Viewstamped replication revisited," Technical Report MIT-CSAIL-TR-2012-021, MIT, 2012.

[3] B. M. Oki and B. H. Liskov, "Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems," presented at the Proceedings of the ACM Symposium on Principles of Distributed Computing, Toronto, Ontario, Canada, 1988.

[4] M. Poke and T. Hoefler, "DARE: High-Performance State Machine Replication on RDMA Networks," in *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing*, Portland, Oregon, USA, 2015, 2749267, pp. 107-118.

[5] P. Hunt, M. Konar, F. P. Junqueira *et al.*, "ZooKeeper: wait-free coordination for internet-scale systems," in *Proceedings of the USENIX annual technical conference*, Boston, MA, 2010.

[6] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *Proceedings of the ACM symposium on principles of distributed computing*, Portland, Oregon, USA, 2007.

[7] D. Mazieres, "Paxos made practical," Technical Report on http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf, 2007.

MODULE *FLAIR*

Formal specification for *FLAIR*.

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

Constants

The set of replicas

CONSTANTS *Replicas*

Replicas states.

CONSTANTS *Follower*, *Leader*

Replicas running state

CONSTANTS *ReplicaUpState*, *ReplicaDownState*

CONSTANTS *SwitchIp*, Ip address of the switch
SwitchKGroupsNum, Number of key groups
SwitchStateActive,
SwitchStateInactive

Message Types

CONSTANTS *ClientReadRequest*, Read request type
ClientWriteRequest, Write request type
WriteResponse, Write response type
ReadResponse, read response type
Read request from the switch to a replica
InternalReadRequest,
Write request from the switch to a replica
InternalWriteRequest,
A request from the leader to the replicas to append a *log* entry
AppendEntriesRequest,
An append entry resposne from a replica to the leader
AppendEntriesResponse

The set of possible keys and values in a client request

CONSTANTS *ValueSpace*, *KeySpace*

Special reserved value

CONSTANTS *Nil*

Variables

Replica *vars*

VARIABLE	$state$,	The replica's state (Follower, or *Leader*).
	log,	Log of all committed and uncommitted operations
	$commitIndex$,	The index of the highest committed index in the *log*
	$currentTerm$,	Current term number
	$isActive$,	Is the replica up or down
	$replicaSession$	The latest switch id seen by the leader

$$replicaVars \triangleq \langle state, log, commitIndex, currentTerm, isActive, replicaSession \rangle$$

Leader *vars*. The following variables are used only on leaders:

VARIABLE	$nextIndex$,	The next entry to send to each follower.
	$matchIndex$,	The entry index for which a follower *log* matches the leader *log*. This used to calculate commit index
	$replicaKGroups$	A map from a key hash to the last received sequence number for the associated *KGroup*

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

Switch *vars*

VARIABLES	$switchKGourpArray$,	Array to maintain information about each *KGroup*
	$switchSeqNum$,	The last sequence number assigned by the switch
	$switchTermId$,	The latest term number seen by the switch
	$switchLeaderId$,	Current leader id as seen by the switch
	$session$,	Current switch id nique value for each session
	$switchState$	Is the switch up or down

$$switchVars \triangleq \langle switchKGourpArray, switchState, switchTermId, switchLeaderId, switchSeqNum, session \rangle$$

Messages variables

VARIABLE	$messages$,	messages among replicas
	$msgsClientSwitch$,	messages from the client to the switch
	$msgsReplicasSwitch$	messages from between replicas and the switch

$$msgsVars \triangleq \langle messages, msgsClientSwitch, msgsReplicasSwitch \rangle$$

Proof *vars*, don't appear in implementations

VARIABLES $responsesToClient$

Set of all variables

$vars \triangleq \langle msgsVars,\ replicaVars,\ leaderVars,\ switchVars,\ responsesToClient \rangle$

Helpers

Helper to *Append* an element to a set

$AddToSet(set,\ element) \triangleq set \cup \{element\}$

Helper to remove an element to a set

$RemoveFromSet(set,\ element) \triangleq set \setminus \{element\}$

Helper to return the minimum value from a set,
or undefined if the set is empty.

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

Helper to return the maximum value from a set,
or undefined if the set is empty.

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

Helper to choose a random element from a set

$ChooseRandomly(set) \triangleq \text{CHOOSE } i \in set : \text{TRUE}$

Helper to return the index of the *KGroup* given a hash

$getIndexFromHash(hash) \triangleq (hash \% SwitchKGroupsNum) + 1$

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

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

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

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

Helper to find list of replicas that agree on a *log* index

$$
\begin{aligned}
&AgreeIndex(index,\ logs,\ leaderId) \triangleq \\
&\quad \{leaderId\} \cup \{k \in Replicas : \\
&\qquad\qquad \land\ k \neq leaderId \\
&\qquad\qquad \land\ Len(logs[k]) \geq index \\
&\qquad\qquad \land\ logs[k][index] = logs[leaderId][index]\}
\end{aligned}
$$

Helper to return all *log* entries for specific key

$entriesForKey(s,\ key) \triangleq \{j \in \text{DOMAIN } log[s] : \land\ log[s][j].key = key\}$

return the max value of a set or *Nil* if the set is empty

$$
\begin{aligned}
indexOfLastEntry(entries) \triangleq\ & \text{IF } Cardinality(entries) > 0 \\
& \text{THEN } Max(entries) \text{ ELSE } Nil
\end{aligned}
$$

helper function to get the *ID* of the leader
if there is a leader that is up, this function returns its *ID*
if there is no currect leader, this function finds the next leader based on raft criteria and returns its *ID*

$$
\begin{array}{l}
getLeaderId \triangleq \\
\quad \text{LET } leadersList \triangleq \{s \in \text{DOMAIN } state : state[s] = \\
\qquad\qquad\qquad Leader \land isActive[s] = ReplicaUpState\} \\
\qquad runningReplicas \triangleq \{s \in Replicas : isActive[s] = ReplicaUpState\} \\
\qquad replicasWithNonEmptyLog \triangleq \{s \in runningReplicas : Len(log[s]) > 0\} \\
\qquad latestTerm \triangleq Max(\{log[s][Len(log[s])].term : \\
\qquad\qquad\qquad s \in replicasWithNonEmptyLog\}) \\
\qquad replicasWithLatestTerm \triangleq \{s \in replicasWithNonEmptyLog : \\
\qquad\qquad\qquad log[s][Len(log[s])].term = latestTerm\} \\
\qquad lengthOfLargestLog \triangleq Max(\{Len(log[s]) : s \in replicasWithLatestTerm\}) \\
\qquad replicasWithLongestLog \triangleq \{s \in replicasWithLatestTerm : \\
\qquad\qquad\qquad Len(log[s]) = lengthOfLargestLog\} \\
\qquad newLeaderId \triangleq \text{IF } Cardinality(replicasWithLongestLog) \geq 1 \\
\qquad\qquad \text{THEN CHOOSE } i \in replicasWithLongestLog : \text{TRUE} \\
\qquad\qquad \text{ELSE IF } Cardinality(runningReplicas) > 0 \\
\qquad\qquad \text{THEN CHOOSE } i \in runningReplicas : \text{TRUE} \\
\qquad\qquad \text{ELSE } Nil \\
\quad \text{IN IF } Cardinality(leadersList) > 0 \\
\quad\quad \text{THEN CHOOSE } s \in leadersList : \text{TRUE} \\
\quad\quad \text{ELSE } newLeaderId
\end{array}
$$

Helper to add a message to a set of messages.

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

Helper to remove a message from a set of messages.
Used when a replica is done processing a *replicaVarsmessage*.

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

Messages schemes and data structures

$$
\begin{array}{l}
createClinetReadRequest(key) \triangleq \\
\quad [mtype \mapsto ClientReadRequest, \\
\quad\ mkey \mapsto key, \\
\quad\ mhash \mapsto key]
\end{array}
$$

$$
\begin{array}{l}
createClinetWriteRequest(key, value) \triangleq \\
\quad [mtype \mapsto ClientWriteRequest, \\
\quad\ mkey \mapsto key, \\
\quad\ mvalue \mapsto value, \\
\quad\ mhash \mapsto key]
\end{array}
$$

When the switch forwards a read request, it adds:
1 − *logIndex*: The replica should *executereplicaVars* this index
before serving the request
2 − *seqNum*: Which is used to ensure the linearizability
of the request response

$$
\begin{array}{l}
createInternalReadRequest(msg, KGroup, forwardTo) \triangleq \\
\quad [mtype \mapsto InternalReadRequest, \\
\quad\ mkey \mapsto msg.mkey, \\
\quad\ mhash \mapsto msg.mhash, \\
\quad\ msession \mapsto session, \\
\quad\ mterm \mapsto switchTermId, \\
\quad\ mleaderId \mapsto switchLeaderId, \\
\quad\ mlogIndex \mapsto KGroup.logIndex, \\
\quad\ mkGroupSeqNum \mapsto KGroup.seqNum, \\
\quad\ msource \mapsto SwitchIp, \\
\quad\ mdest \mapsto forwardTo]
\end{array}
$$

When the switch forwards a write request, it adds a
sequence number that is used to order write requests.
The leader processes the requests in order.

$$
\begin{array}{l}
createInternalWriteRequest(msg, KGroup) \triangleq \\
\quad [mtype \mapsto InternalWriteRequest, \\
\quad\ mkey \mapsto msg.mkey, \\
\quad\ mvalue \mapsto msg.mvalue, \\
\quad\ mhash \mapsto msg.mhash, \\
\quad\ msession \mapsto session, \\
\quad\ mterm \mapsto switchTermId, \\
\quad\ mleaderId \mapsto switchLeaderId, \\
\quad\ mkGroupSeqNum \mapsto KGroup.seqNum, \\
\quad\ msource \mapsto SwitchIp, \\
\quad\ mdest \mapsto switchLeaderId]
\end{array}
$$

$$
\begin{array}{l}
createReadResponse(m, i, leaderId, value, status, logIndex) \triangleq \\
\quad [mtype \mapsto ReadResponse, \\
\quad\ mkey \mapsto m.mkey, \\
\quad\ mvalue \mapsto value, \\
\quad\ mhash \mapsto m.mhash, \\
\quad\ mstatus \mapsto status, \\
\quad\ mlogIndex \mapsto logIndex, \\
\quad\ mkGroupSeqNum \mapsto m.mkGroupSeqNum, \\
\quad\ mterm \mapsto currentTerm[i], \\
\quad\ mleaderId \mapsto leaderId, \\
\quad\ mallLogs \mapsto log, \quad \text{for correctness check only} \\
\quad\ mcommitIndex \mapsto commitIndex, \\
\quad\ msession \mapsto m.msession,
\end{array}
$$

$$
\begin{array}{ll}
\quad msource & \mapsto i, \\
\quad mdest & \mapsto SwitchIp]
\end{array}
$$

$$
\begin{array}{l}
createWriteResponse(i, logEntry, index, status, replicaIds) \triangleq \\
\quad \begin{array}{ll}
[mtype & \mapsto WriteResponse, \\
\ mkey & \mapsto logEntry.key, \\
\ mvalue & \mapsto logEntry.value, \\
\ mhash & \mapsto logEntry.hash, \\
\ mstatus & \mapsto status, \\
\ mlogIndex & \mapsto index, \\
\ mkGroupSeqNum & \mapsto logEntry.seqNum, \\
\ msession & \mapsto logEntry.switchId, \\
\ mreplicaIds & \mapsto replicaIds, \\
\ mterm & \mapsto currentTerm[i], \\
\ mallLogs & \mapsto log, \ \text{for correctness check only} \\
\ mcommitIndex & \mapsto commitIndex, \\
\ msource & \mapsto i, \\
\ mdest & \mapsto SwitchIp]
\end{array}
\end{array}
$$

$$
\begin{array}{l}
createKGroup(leaderAcked, replicasIds, seqNum, logIndex) \triangleq \\
\quad [leaderAcked \mapsto leaderAcked, \\
\quad\ replicasIds \mapsto replicasIds, \\
\quad\ seqNum \mapsto seqNum, \\
\quad\ logIndex \mapsto logIndex]
\end{array}
$$

$$
\begin{array}{l}
createLogEntry(i, msg) \triangleq \\
\quad \begin{array}{ll}
[term & \mapsto currentTerm[i], \\
\ key & \mapsto msg.mkey, \\
\ value & \mapsto msg.mvalue, \\
\ hash & \mapsto msg.mhash, \\
\ seqNum & \mapsto msg.mkGroupSeqNum, \\
\ switchId & \mapsto msg.msession]
\end{array}
\end{array}
$$

$$
\begin{array}{l}
createResHistoryEntry(msg, tag) \triangleq \\
\quad \begin{array}{ll}
[msg & \mapsto msg, \\
\ switchKGroupEntry & \mapsto switchKGourpArray[getIndexFromHash(msg.mhash)], \\
\ tag & \mapsto tag]
\end{array}
\end{array}
$$

Variables initialization

$InitSwitchVars \triangleq$

Initially the switch is inactive and

KGroupArray is stable

$\land$ $switchKGourpArray = [i \in 1 \,..\, SwitchKGroupsNum \mapsto createKGroup(\text{TRUE}, \{\}, Nil, Nil)]$
$\land$ $switchState = SwitchStateInactive$
$\land$ $switchTermId = 0$
$\land$ $switchLeaderId = Nil$
$\land$ $switchSeqNum = 0$
$\land$ $session = 0$

$InitMsgsSets \triangleq$
$\land$ $msgsClientSwitch = \{\}$
$\land$ $msgsReplicasSwitch = \{\}$
$\land$ $messages = \{\}$

$InitReplicaVars \triangleq$
$\land$ $currentTerm = [i \in Replicas \mapsto 1]$
$\land$ $state = [i \in Replicas \mapsto Follower]$
$\land$ $isActive = [i \in Replicas \mapsto ReplicaUpState]$
$\land$ $replicaSession = [i \in Replicas \mapsto 0]$
$\land$ $log = [i \in Replicas \mapsto \langle\rangle]$
$\land$ $commitIndex = [i \in Replicas \mapsto 0]$

$InitLeaderVars \triangleq$
$\land$ $nextIndex = [i \in Replicas \mapsto [j \in Replicas \mapsto 1]]$
$\land$ $matchIndex = [i \in Replicas \mapsto [j \in Replicas \mapsto 0]]$
$\land$ $replicaKGroups = [i \in Replicas \mapsto [j \in 1 \,..\, SwitchKGroupsNum \mapsto 0]]$

$Init \triangleq$
$\land$ $InitReplicaVars$
$\land$ $InitLeaderVars$
$\land$ $InitSwitchVars$
$\land$ $InitMsgsSets$
Needed to prove safety
$\land$ $responsesToClient = \{\}$

Variables actions

Client actions

client sends a read request to read key k

$IssueReadRequest(key) \triangleq$
$\land$ LET $request \triangleq createClinetReadRequest(key)$
IN $msgsClientSwitch' = AddToSet(msgsClientSwitch, request)$
$\land$ UNCHANGED $\langle messages, replicaVars, leaderVars, switchVars, msgsReplicasSwitch, responsesToClient\rangle$

client issues a write request to update key k

$$
\begin{array}{l}
\mathit{IssueWriteRequest}(\mathit{key}, \mathit{value}) \triangleq \\
\quad \land \textsc{let } \mathit{request} \triangleq \mathit{createClinetWriteRequest}(\mathit{key}, \mathit{value}) \\
\quad \quad \textsc{in } \mathit{msgsClientSwitch}' = \mathit{AddToSet}(\mathit{msgsClientSwitch}, \mathit{request}) \\
\quad \land \textsc{unchanged } \langle \mathit{messages}, \mathit{replicaVars}, \mathit{leaderVars}, \\
\quad \quad \quad \mathit{switchVars}, \mathit{msgsReplicasSwitch}, \mathit{responsesToClient} \rangle
\end{array}
$$

Switch actions

Switch state changes from Active to inactive

$$
\begin{array}{l}
\mathit{switchFails} \triangleq \\
\quad \land \mathit{switchState} = \mathit{SwitchStateActive} \\
\quad \land \mathit{switchState}' = \mathit{SwitchStateInactive} \\
\quad \land \textsc{unchanged } \langle \mathit{replicaVars}, \mathit{leaderVars}, \mathit{msgsVars}, \mathit{responsesToClient}, \\
\quad \quad \quad \mathit{switchSeqNum}, \mathit{session}, \mathit{switchKGourpArray}, \\
\quad \quad \quad \mathit{switchLeaderId}, \mathit{switchTermId} \rangle
\end{array}
$$

Switch handles a read request from a client.
The request has a hash that is mapped to a
KGroup. If the *KGroup* is stable, the request will
be forwarded to one of the replcias, otherwise it
will be forwarded to the leader.

$$
\begin{array}{l}
\mathit{SwitchHandleCleintRead}(\mathit{msg}) \triangleq \\
\quad \textsc{let } \mathit{kGroup} \triangleq \mathit{switchKGourpArray}[\mathit{getIndexFromHash}(\mathit{msg.mhash})] \\
\quad \quad \mathit{forwardTo} \triangleq \textsc{if } \mathit{kGroup.leaderAcked} \land \\
\quad \quad \quad \quad \quad \mathit{kGroup.seqNum} \neq \mathit{Nil} \\
\quad \quad \quad \textsc{then } \mathit{ChooseRandomly}(\{x \in \mathit{kGroup.replicasIds} : \\
\quad \quad \quad \quad \quad x \neq \mathit{switchLeaderId}\}) \\
\quad \quad \quad \textsc{else } \textsc{if } \mathit{kGroup.leaderAcked} \land \\
\quad \quad \quad \quad \quad \mathit{kGroup.seqNum} = \mathit{Nil} \\
\quad \quad \quad \textsc{then } \mathit{switchLeaderId} \\
\quad \quad \quad \textsc{else } \mathit{switchLeaderId} \\
\quad \quad \mathit{internalMsg} \triangleq \mathit{createInternalReadRequest}(\mathit{msg}, \mathit{kGroup}, \mathit{forwardTo}) \\
\quad \textsc{in} \\
\quad \quad \land \mathit{msgsReplicasSwitch}' = \mathit{AddToSet}(\mathit{msgsReplicasSwitch}, \mathit{internalMsg}) \\
\quad \quad \land \textsc{unchanged } \langle \mathit{replicaVars}, \mathit{leaderVars}, \mathit{switchVars}, \\
\quad \quad \quad \quad \mathit{responsesToClient}, \mathit{messages}, \mathit{msgsClientSwitch} \rangle
\end{array}
$$

Switch handles a write request from a client

$$
\begin{array}{l}
\mathit{SwitchHandleCleintWrite}(\mathit{msg}) \triangleq
\end{array}
$$

Mark the *KGroup* associated with the key as unstable

$$
\quad \textsc{let } \mathit{kGroup} \triangleq \mathit{switchKGourpArray}[\mathit{getIndexFromHash}(\mathit{msg.mhash})]
$$

$$
\begin{array}{l}
\quad\quad updatedKGroup \triangleq [kGroup \text{ EXCEPT } !.leaderAcked = \text{FALSE},\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad !.seqNum = switchSeqNum + 1,\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad !.replicasIds = \{\},\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad !.logIndex = Nil]\\
\quad\quad internalMsg \triangleq createInternalWriteRequest(msg, updatedKGroup)\\
\quad \text{IN}\\
\quad\quad \land msgsReplicasSwitch' = AddToSet(msgsReplicasSwitch, internalMsg)\\
\quad\quad \land switchKGourpArray' = [switchKGourpArray \text{ EXCEPT}\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad ![getIndexFromHash(msg.mhash)] = updatedKGroup]\\
\quad\quad \land switchSeqNum' = switchSeqNum + 1 \quad \text{Increments the sequence number}\\
\quad\quad \land \text{UNCHANGED } \langle replicaVars, leaderVars, responsesToClient,\\
\quad\quad\quad\quad\quad\quad messages, msgsClientSwitch, switchState,\\
\quad\quad\quad\quad\quad\quad switchTermId, switchLeaderId, session\rangle
\end{array}
$$

Switch receives a read or write request from a client

$$
\begin{array}{l}
SwitchReceiveFromClient \triangleq\\
\quad \land switchState = SwitchStateActive\\
\quad \land Cardinality(msgsClientSwitch) > 0\\
\quad \land \text{LET } msg \triangleq ChooseRandomly(msgsClientSwitch)\\
\quad\quad\quad\quad type \triangleq msg.mtype\\
\quad\quad \text{IN} \quad \lor \land type = ClientReadRequest\\
\quad\quad\quad\quad\quad \land SwitchHandleCleintRead(msg)\\
\quad\quad\quad\quad \lor \land type = ClientWriteRequest\\
\quad\quad\quad\quad\quad \land SwitchHandleCleintWrite(msg)
\end{array}
$$

Switch handles a read response

$$
\begin{array}{l}
SwitchHandleReadResponse(msg) \triangleq\\
\quad \land \lor \land msg.msource \neq switchLeaderId \quad msg \text{ is from follower}\\
\quad\quad\quad \land msg.mterm = switchTermId \quad msg.term = \text{ switch term}\\
\quad\quad\quad \land msg.mstatus = \text{TRUE} \quad \text{The operation was succeeded}\\
\quad\quad\quad \text{Map the key to a } KGroup \text{ based on hash.}\\
\quad\quad\quad \text{The response will be sent to teh client if}\\
\quad\quad\quad 1 - msg.seqNum = KGroup.seqNum\\
\quad\quad\quad 2 - KGroup \text{ is stable}\\
\quad\quad\quad \text{otherwise the request will be dropped}\\
\quad\quad\quad \land \text{LET } KGroup \triangleq switchKGourpArray[getIndexFromHash(msg.mhash)]\\
\quad\quad\quad\quad\quad isSeqOk \triangleq KGroup.seqNum = msg.mkGroupSeqNum\\
\quad\quad\quad\quad \text{IN} \quad \lor \land isSeqOk\\
\quad\quad\quad\quad\quad\quad \land KGroup.leaderAcked\\
\quad\quad\quad\quad\quad\quad \land responsesToClient' = AddToSet(\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad responsesToClient,\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad createResHistoryEntry(msg, Nil))
\end{array}
$$

$\lor \land \neg isSeqOk$
$\quad \land$ UNCHANGED $\langle responsesToClient \rangle$
$\lor \land \neg KGroup.leaderAcked$
$\quad \land isSeqOk$
$\quad \land$ UNCHANGED $\langle responsesToClient \rangle$

$\lor \land msg.msource \neq switchLeaderId$
$\quad \land msg.mstatus = $ FALSE The key does not exist
$\quad \land$ UNCHANGED $\langle responsesToClient \rangle$

The operation succeeded and the response
is from the current leader. Just forward it to the client

$\lor \land msg.msource = switchLeaderId$
$\quad \land msg.mterm = switchTermId$
$\quad \land msg.mstatus = $ TRUE
$\quad \land responsesToClient' = AddToSet(responsesToClient, createResHistoryEntry(msg, Nil))$

$\lor \land msg.msource = switchLeaderId$
$\quad \land msg.mstatus = $ FALSE The key does not exist
$\quad \land$ UNCHANGED $\langle responsesToClient \rangle$

$\land$ UNCHANGED $\langle replicaVars, leaderVars, switchVars, msgsVars \rangle$

Switch handles a write response from a client.
If the *res.seqNum* equals the *KGroup.seqNum*, the switch
marks the *KGroup* as stable and forwards the response to
the client. If the $res.seqNum < KGroup.seqNum$, the switch
will not change the status of the *KGroup*, and will just
forward the response to the clinet

$SwitchHandleWriteResponse(msg) \triangleq$
$\land \lor \land msg.msource = switchLeaderId$ Message from the leader
$\quad \land msg.mterm = switchTermId$ $Message.term = switch.term$
$\quad \land msg.mstatus = $ TRUE The operation succeeded
$\quad \land$ LET $KGroup \triangleq switchKGourpArray[getIndexFromHash(msg.mhash)]$
$\qquad isSeqOk \triangleq KGroup.seqNum = msg.mkGroupSeqNum$
$\qquad updatedKGroup \triangleq [KGroup$ EXCEPT
$\qquad\qquad !.leaderAcked = $ TRUE,
$\qquad\qquad !.replicasIds = msg.mreplicaIds,$
$\qquad\qquad !.logIndex = msg.mlogIndex]$

$\qquad msg.seqNum = KGroup.seqNum$

$\quad$ IN $\lor \land isSeqOk$

$\qquad$ Mark the *KGroup* as stable

$\qquad \land switchKGourpArray' = [switchKGourpArray$ EXCEPT $![getIndexFromHash(msg.mhash)] = updatedKGroup]$
$\qquad \land responsesToClient' = AddToSet(responsesToClient, createResHistoryEntry(msg, Nil))$

$msg.seqNum ! = KGroup.seqNum$

$$
\begin{array}{l}
\qquad\qquad \lor \land \neg isSeqOk \\
\qquad\qquad\quad \land responsesToClient' = AddToSet(responsesToClient, \\
\qquad\qquad\qquad\qquad createResHistoryEntry(msg, Nil)) \\
\qquad\qquad\quad \land \text{UNCHANGED } \langle switchKGourpArray \rangle \\
\quad \lor \land msg.msource \neq switchLeaderId \\
\quad\quad \land \text{UNCHANGED } \langle switchKGourpArray, responsesToClient \rangle \\
\quad \lor \land msg.mstatus = \text{FALSE} \\
\quad\quad \land \text{UNCHANGED } \langle switchKGourpArray, responsesToClient \rangle \\
\land \text{UNCHANGED } \langle replicaVars, leaderVars, msgsVars, \\
\qquad switchSeqNum, session, switchLeaderId, \\
\qquad switchTermId, switchState \rangle
\end{array}
$$

Switch receives a message from a replica

$SwitchReceiveFromReplica(msg) \triangleq$

msg term id is larger than switch term id

$\Rightarrow$ switch stops processing request by setting its status to inactive

$$
\begin{array}{l}
\lor \land switchState = SwitchStateActive \\
\quad \land msg.msession = session \\
\quad \land msg.mterm > switchTermId \\
\quad \land switchState' = SwitchStateInactive \\
\quad \land \text{UNCHANGED } \langle replicaVars, leaderVars, msgsVars, \\
\qquad\qquad msgsClientSwitch, switchVars, responsesToClient \rangle
\end{array}
$$

msg is coming from an old leader

$\Rightarrow$ switch just ignore the message

$$
\begin{array}{l}
\lor \land switchState = SwitchStateActive \\
\quad \land msg.msession = session \\
\quad \land msg.mterm < switchTermId \\
\quad \land \text{UNCHANGED } \langle replicaVars, leaderVars, msgsVars, \\
\qquad\qquad responsesToClient, switchVars \rangle
\end{array}
$$

$msg.switchId$ does not match $switchId$

$\Rightarrow$ switch just ignore the message

$$
\begin{array}{l}
\lor \land switchState = SwitchStateActive \\
\quad \land msg.msession \neq session \\
\quad \land \text{UNCHANGED } \langle replicaVars, leaderVars, msgsVars, \\
\qquad\qquad responsesToClient, switchVars \rangle
\end{array}
$$

Switch is active and the read response passes the safety check

$\Rightarrow$ switch processes the read response

$$
\begin{array}{l}
\lor \land switchState = SwitchStateActive \\
\quad \land msg.msession = session \\
\quad \land msg.mtype = ReadResponse \\
\quad \land SwitchHandleReadResponse(msg)
\end{array}
$$

Switch is active and the write response passes the safety check
$\Rightarrow$ switch processes the write response

$$
\begin{array}{l}
\lor \ \land \mathit{switchState} = \mathit{SwitchStateActive} \\
\quad \land \mathit{msg.msession} = \mathit{session} \\
\quad \land \mathit{msg.mtype} = \mathit{WriteResponse} \\
\quad \land \mathit{SwitchHandleWriteResponse}(\mathit{msg})
\end{array}
$$

Replica actions

Replica i fails and stops processing *msgs*.
It loses everything but its *currentTerm* and *log*.

$$
\begin{array}{l}
\mathit{Stop}(i) \triangleq \\
\quad \land \mathit{isActive}[i] = \mathit{ReplicaUpState} \\
\quad \land \mathit{isActive}' \ = [\mathit{isActive} \ \text{EXCEPT} \ ![i] = \mathit{ReplicaDownState}] \\
\quad \land \text{UNCHANGED} \ \langle \mathit{leaderVars}, \mathit{msgsVars}, \mathit{switchVars}, \mathit{responsesToClient}, \\
\qquad\qquad\qquad \mathit{currentTerm}, \mathit{log}, \ \mathit{commitIndex}, \\
\qquad\qquad\qquad \mathit{state}, \mathit{replicaSession} \rangle
\end{array}
$$

Replica i becomes active. The replica stars as follower

$$
\begin{array}{l}
\mathit{Start}(i) \triangleq \\
\quad \land \mathit{isActive}[i] \quad = \mathit{ReplicaDownState} \\
\quad \land \mathit{isActive}' \quad = [\mathit{isActive} \ \text{EXCEPT} \ ![i] = \mathit{ReplicaUpState}] \\
\quad \land \mathit{state}' \quad = [\mathit{state} \ \text{EXCEPT} \ ![i] = \mathit{Follower}] \\
\quad \land \mathit{nextIndex}' \quad = [\mathit{nextIndex} \ \text{EXCEPT} \ ![i] = [j \in \mathit{Replicas} \mapsto 1]] \\
\quad \land \mathit{matchIndex}' \ = [\mathit{matchIndex} \ \text{EXCEPT} \ ![i] = [j \in \mathit{Replicas} \mapsto 0]] \\
\quad \land \mathit{commitIndex}' = [\mathit{commitIndex} \ \text{EXCEPT} \ ![i] = 0] \\
\quad \land \mathit{replicaKGroups}' = [\mathit{replicaKGroups} \ \text{EXCEPT} \ ![i] = \\
\qquad\qquad\qquad\qquad [j \in 1 \mathrel{..} \mathit{SwitchKGroupsNum} \mapsto 0]] \\
\quad \land \text{UNCHANGED} \ \langle \mathit{msgsVars}, \mathit{switchVars}, \mathit{responsesToClient}, \\
\qquad\qquad\qquad \mathit{currentTerm}, \mathit{log}, \mathit{replicaSession} \rangle
\end{array}
$$

Select a new leader if non of the replicas is a leader
This helper selects the new leader based on raft criteria.
That is, the node with the *log* with the highest term id and
longest *log*, if mutiple nodes have the same *log* id.

$$
\begin{array}{l}
\mathit{ElectLeader} \triangleq \\
\quad \land \text{LET} \ \ \mathit{runningReplicas} \triangleq \{s \in \mathit{Replicas} : \mathit{isActive}[s] = \mathit{ReplicaUpState}\} \\
\qquad\qquad \mathit{replicasWithNonEmptyLog} \triangleq \{s \in \mathit{runningReplicas} : \mathit{Len}(\mathit{log}[s]) > 0\} \\
\qquad\qquad \mathit{latestTerm} \triangleq \mathit{Max}(\{\mathit{log}[s][\mathit{Len}(\mathit{log}[s])].\mathit{term} : \\
\qquad\qquad\qquad\qquad s \in \mathit{replicasWithNonEmptyLog}\}) \\
\qquad\qquad \mathit{replicasWithLatestTerm} \triangleq \{s \in \mathit{replicasWithNonEmptyLog} : \\
\qquad\qquad\qquad\qquad \mathit{log}[s][\mathit{Len}(\mathit{log}[s])].\mathit{term} = \mathit{latestTerm}\} \\
\qquad\qquad \mathit{lengthOfLargestLog} \triangleq \mathit{Max}(\{\mathit{Len}(\mathit{log}[s]) : \\
\qquad\qquad\qquad\qquad s \in \mathit{replicasWithLatestTerm}\})
\end{array}
$$

```
replicasWithLongestLog ≜ {s ∈ replicasWithLatestTerm :
                            Len(log[s]) = lengthOfLargestLog}
newLeaderId ≜ IF   Cardinality(replicasWithLongestLog) ≥ 1
                THEN CHOOSE i ∈ replicasWithLongestLog : TRUE
                ELSE IF Cardinality(runningReplicas) > 0
                THEN CHOOSE i ∈ runningReplicas : TRUE
                ELSE Nil
newTerm     ≜ IF newLeaderId ≠ Nil
                THEN currentTerm[newLeaderId] + 1 ELSE Nil
majority    ≜ IF newLeaderId ≠ Nil
                THEN CHOOSE g ∈ Quorum : newLeaderId ∈ g
                ELSE Nil
newCurrentTerm ≜ [j ∈ Replicas ↦ IF j ∈ majority
                                    THEN newTerm
                                    ELSE currentTerm[j]]
IN   ∧ Cardinality(runningReplicas) * 2 > Cardinality(Replicas)
     ∧ ∀ i ∈ runningReplicas : state[i] ∈ {Follower}
     ∧ state'        = [state EXCEPT ![newLeaderId] = Leader]
     ∧ nextIndex'    = [nextIndex EXCEPT ![newLeaderId] =
                         [j ∈ Replicas ↦ Len(log[newLeaderId]) + 1]]
     ∧ matchIndex'   = [matchIndex EXCEPT ![newLeaderId] =
                         [j ∈ Replicas ↦ 0]]
     ∧ currentTerm' = newCurrentTerm
∧ UNCHANGED ⟨switchVars, msgsVars, responsesToClient,
              replicaKGroups, commitIndex, log,
              isActive, replicaSession⟩
```

leader i populates the switch *KGroup* array with information about unstable *KGroups*

```
fillSwitchKGroup(i) ≜
  LET startIndex ≜ 1
    endIndex    ≜ Len(log[i])
```

Scan the *log* and map each key in the *log* to its associated *KGroup*

```
    keysToKGroups ≜ [j ∈ startIndex . . endIndex ↦
                        getIndexFromHash(log[i][j].hash)]
```

For each *KGroup*, find the last *log* entry that should be used to update the switch *KGroup*

```
    mapLogEntryToKGroupIndex ≜
      [j ∈ 1 . . SwitchKGroupsNum ↦
       IF Cardinality({k ∈ DOMAIN keysToKGroups : keysToKGroups[k] = j}) > 0
        THEN Max({k ∈ DOMAIN keysToKGroups : keysToKGroups[k] = j})
        ELSE Nil]
```

A boolean array that indicates whether a *log* entry is committed or not

$leaderAckedFlag \triangleq [j \in startIndex \,..\, endIndex \mapsto j \leq commitIndex[i]]$

Get the set of replicas that acknowleged each *log* entry

$keysToReplicas \triangleq [j \in startIndex \,..\, endIndex \mapsto \text{IF } leaderAckedFlag[j] \text{ THEN } AgreeIndex(j, log, i) \text{ ELSE } \{i\}]$

IN Update the switch *KGroup* Array to match the leader's *KGroup*. If the leader do not have any writes for a *KGroup*, then the *KGroup* is stable

$\land\ switchKGourpArray' =$
$[j \in 1 \,..\, SwitchKGroupsNum \mapsto$
$\text{IF } mapLogEntryToKGroupIndex[j] \neq Nil$
$\text{THEN } createKGroup(leaderAckedFlag[mapLogEntryToKGroupIndex[j]], keysToReplicas[mapLogEntryToKGroupIndex[j]], 0, mapLogEntryToKGroupIndex[j])$
$\text{ELSE } createKGroup(\text{TRUE}, Replicas, Nil, Nil)]$

Leader i activates the switch

$LeaderActivateSwitch(i) \triangleq$
$\land\ state[i] = Leader$ Replica is the leader
$\land\ isActive[i] = ReplicaUpState$ Replica is active
$\land\ switchState = SwitchStateInactive$ Switch is inactive
$\land\ replicaSession' = [replicaSession \text{ EXCEPT } ![i] = session + 1]$
$\land\ session' = session + 1$ Update the replica and switch sessions
$\land\ switchTermId' = currentTerm[i]$ Update switch term number
$\land\ switchLeaderId' = i$ Update leader id of the switch
$\land\ switchSeqNum' = 0$ Reset the sequence number
$\land\ fillSwitchKGroup(i)$ Populate switch *KGroup* array
$\land\ switchState' = SwitchStateActive$ activate the switch
$\land\ \text{UNCHANGED } \langle leaderVars, msgsVars, responsesToClient, state, log, commitIndex, currentTerm, isActive\rangle$

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

$UpdateTerm(i, j, m) \triangleq$
$\land\ m.mterm > currentTerm[i]$
$\land\ currentTerm' = [currentTerm \text{ EXCEPT } ![i] = m.mterm]$
$\land\ state' = [state \text{ EXCEPT } ![i] = Follower]$
messages is unchanged so m can be processed further.

$\land$ UNCHANGED $\langle switchVars, msgsVars, leaderVars,$
$responsesToClient, isActive, log,$
$commitIndex, replicaSession\rangle$

Responses with stale terms are ignored.

$DropStaleResponse(i, j, m) \triangleq$
$\quad \land m.mterm < currentTerm[i]$
$\quad \land$ UNCHANGED $\langle replicaVars, leaderVars, switchVars,$
$responsesToClient, msgsVars\rangle$

Replica i receives a read request (m) from a client.

$ReplicaReceiveReadRequest(m, i) \triangleq$

The request received by the leader

$\lor \land state[i] = Leader$

Check the message term numebr

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

Get the index of the last committed entry

$\land$ LET $committedEntries \triangleq \{j \in$ DOMAIN $log[i] : \land log[i][j].key = m.mkey$
$\land j \leq commitIndex[i]\}$
$lastEntryIndex \triangleq$ IF $Cardinality(committedEntries) > 0$
THEN $Max(committedEntries)$ ELSE Nil
$success \triangleq$ IF $lastEntryIndex = Nil$ THEN FALSE ELSE TRUE
$value \triangleq$ IF $success$ THEN $log[i][lastEntryIndex].value$ ELSE Nil
IN $\land msgsReplicasSwitch' = AddToSet(msgsReplicasSwitch,$
$createReadResponse(m, i, getLeaderId,$
$value, success, lastEntryIndex))$
$\land$ UNCHANGED $\langle replicaVars, leaderVars, switchVars,$
$msgsClientSwitch, messages,$
$responsesToClient\rangle$

The request received by a follower and $msg.mlogIndex > 0$
$i.e$, the switch processed a write associated with the same
$KGroup$ of the key

$\lor \land state[i] = Follower$
$\land m.mterm = currentTerm[i]$ $msg.term =$ replica term
$\land m.mlogIndex \leq Len(log[i])$ The replica has the index
$\land m.mlogIndex > 0$ Switch $Kgroup$ entry is not empty

Get the last committed log entry for the requested key

$\land$ LET $logEntriesForKey \triangleq entriesForKey(i, m.mkey)$
$filteredEntries \triangleq \{j \in logEntriesForKey : j \leq m.mlogIndex\}$
$lastEntryIndex \triangleq$ IF $Cardinality(filteredEntries) > 0$
THEN $Max(filteredEntries)$ ELSE Nil
$requestedEntry \triangleq$ IF $Cardinality(filteredEntries) > 0$
THEN $log[i][lastEntryIndex]$ ELSE Nil

```
            isCommitted ≜ m.mlogIndex ≤ commitIndex[i]
            success ≜ IF Cardinality(filteredEntries) > 0
                          THEN requestedEntry.key = m.mkey ELSE FALSE
            value ≜ IF success THEN requestedEntry.value ELSE Nil
      IN  ∧ msgsReplicasSwitch' = AddToSet(msgsReplicasSwitch,
                                    createReadResponse(m, i, getLeaderId,
                                        value, success, lastEntryIndex))
          ∧ ∨ ∧ isCommitted
              ∧ UNCHANGED ⟨commitIndex⟩
            ∨ ∧ ¬isCommitted
              ∧ success
              ∧ commitIndex' = [commitIndex EXCEPT ![i] = m.mlogIndex]
   ∧ UNCHANGED ⟨leaderVars, switchVars, responsesToClient,
                log, state, currentTerm, isActive,
                msgsClientSwitch, messages, replicaSession⟩

   The request received by a follower and msg.mlogIndex = −1.
   i.e., the switch did not process any write associated
∨ ∧ state[i] = Follower
  ∧ m.mterm = currentTerm[i]   msg.term = replica term
  ∧ m.mlogIndex = Nil          Switch Kgroup entry is empty
    Get the last committed log entry for the requested key
  ∧ LET committedEntries ≜ {j ∈ DOMAIN log[i] : ∧ log[i][j].key = m.mkey
                                                ∧ j ≤ commitIndex[i]}
        lastEntryIndex ≜ IF Cardinality(committedEntries) > 0
                            THEN Max(committedEntries) ELSE Nil
        success ≜ IF lastEntryIndex = Nil THEN FALSE ELSE TRUE
        value ≜ IF success THEN log[i][lastEntryIndex].value ELSE Nil
    IN  ∧ msgsReplicasSwitch' = AddToSet(msgsReplicasSwitch,
                                  createReadResponse(m, i,
                                      getLeaderId, value, success,
                                      lastEntryIndex))

  ∧ UNCHANGED ⟨replicaVars, leaderVars, switchVars,
               responsesToClient, msgsClientSwitch,
               messages⟩


Leader i receives a write request.
ReplicaReceiveWriteRequest(m, i) ≜
    Safety checks
  ∨ ∧ m.mterm = currentTerm[i]   msg.term = replica.term
    ∧ m.mleaderId = i    The leaderId field in the message is the replica
    ∧ state[i] = Leader          The replica is the leader
    ∧ m.msession = replicaSession[i]   msg.session = replica.session
      Get the highest sequence number received fot his KGroup
```

```
∧ LET latestSeenSeqNum ≜ replicaKGroups[i][getIndexFromHash(m.mhash)]
      entry ≜ createLogEntry(i, m)
      newLog ≜ Append(log[i], entry)
       msg.seqNum > KGroup.seqNum
  IN   ∨ ∧ m.mkGroupSeqNum > latestSeenSeqNum
           Update the KGroup seqNum
         ∧ replicaKGroups′ = [replicaKGroups EXCEPT ![i] =
                                  [@ EXCEPT ![getIndexFromHash(m.mhash)] =
                                  m.mkGroupSeqNum]]
         ∧ log′ = [log EXCEPT ![i] = newLog]  Append to log
         ∧ UNCHANGED ⟨switchVars, msgsVars, responsesToClient,
                        nextIndex, matchIndex, commitIndex,
                        replicaSession, state, currentTerm, isActive⟩
       ∨ ∧ m.mkGroupSeqNum ≤ latestSeenSeqNum
         ∧ UNCHANGED ⟨replicaVars, leaderVars, switchVars,
                        responsesToClient, msgsVars⟩
∨ ∧ ∨ m.mterm     ≠ currentTerm[i]
    ∨ m.mleaderId ≠ i
    ∨ state[i]    ≠ Leader
    ∨ m.msession ≠ replicaSession[i]
  ∧ UNCHANGED ⟨replicaVars, leaderVars, switchVars,
                 responsesToClient, msgsVars⟩
```

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

```
AppendEntries(i, j) ≜
    ∧ i ≠ j  Avoid sending to itself
    ∧ state[i] = Leader  The sender is the leader
    ∧ isActive[i] = ReplicaUpState  The sender is active
     Get the index of the next entry that must
     be sent to the follower j
    ∧ LET prevLogIndex ≜ nextIndex[i][j] − 1
          prevLogTerm ≜ IF prevLogIndex > 0 THEN
                             log[i][prevLogIndex].term
                         ELSE
                             0
           Send up to 1 entry, constrained by the end of the log.
          lastEntry ≜ Min({Len(log[i]), nextIndex[i][j]})
          entries ≜ SubSeq(log[i], nextIndex[i][j], lastEntry)
          m ≜        [mtype          ↦ AppendEntriesRequest,
                       mterm          ↦ currentTerm[i],
                       mprevLogIndex  ↦ prevLogIndex,
                       mprevLogTerm   ↦ prevLogTerm,
```

$$
\begin{array}{ll}
mentries & \mapsto entries, \\
\multicolumn{2}{l}{\text{mlog is used as a history variable for the proof.}} \\
\multicolumn{2}{l}{\text{It would not exist in a real implementation.}} \\
mlog & \mapsto log[i], \\
mcommitIndex & \mapsto Min(\{commitIndex[i], lastEntry\}), \\
msource & \mapsto i, \\
mdest & \mapsto j]
\end{array}
$$

$$
\begin{array}{l}
\quad \text{IN} \quad \land Len(entries) > 0 \\
\qquad\quad\;\; \land Send(m) \\
\land \text{UNCHANGED } \langle replicaVars, leaderVars, switchVars, \\
\qquad\qquad\quad msgsClientSwitch, msgsReplicasSwitch, \\
\qquad\qquad\quad responsesToClient\rangle
\end{array}
$$

Leader i advances its *commitIndex*.
This is done as a separate step from handling *AppendEntries* responses,
in part to minimize atomic regions, and in part so that leaders of
single-server clusters are able to mark entries committed.

$$
\begin{array}{l}
AdvanceCommitIndex(i) \triangleq \\
\quad \land state[i] = Leader \\
\quad \land isActive[i] = ReplicaUpState \\
\quad \land \text{LET } \text{The set of replicas that agree up through an index.} \\
\qquad\quad Agree(index) \triangleq \{i\} \cup \{k \in Replicas : \\
\qquad\qquad\qquad\qquad\qquad\qquad matchIndex[i][k] \geq index\} \\
\qquad\quad \text{The maximum indexes for which a quorum agrees} \\
\qquad\quad agreeIndexes \triangleq \{index \in 1 \,..\, Len(log[i]) : \\
\qquad\qquad\qquad\qquad\qquad Agree(index) \in Quorum\} \\
\qquad\quad \text{Entries that are replicated to majorities} \\
\qquad\quad \text{but not yet committed} \\
\qquad\quad IndicesToCommit \triangleq \{index \in agreeIndexes : index > commitIndex[i]\} \\
\qquad\quad majoritiesPerIndex \triangleq [index \in agreeIndexes \mapsto Agree(index)] \\
\qquad\quad \text{New value for } commitIndex'[i] \\
\qquad\quad newCommitIndex \triangleq \\
\qquad\qquad \text{IF } \land agreeIndexes \neq \{\} \\
\qquad\qquad\quad\;\; \land log[i][Max(agreeIndexes)].term = currentTerm[i] \\
\qquad\qquad \text{THEN } Max(agreeIndexes) \\
\qquad\qquad \text{ELSE } commitIndex[i] \\
\quad\;\; \text{IN} \quad \land newCommitIndex > commitIndex[i] \\
\qquad\qquad \text{For each entry that will be committed, send a write} \\
\qquad\qquad \text{response to clients} \\
\qquad\qquad \land \text{LET } indices \triangleq commitIndex[i] + 1 \,..\, newCommitIndex \\
\qquad\qquad\qquad\; msgs \triangleq [x \in indices \mapsto \\
\qquad\qquad\qquad\qquad\qquad createWriteResponse(i, log[i][x]),
\end{array}
$$

x, TRUE, $majoritiesPerIndex[x])]$

$msgsAsSet \triangleq \{createWriteResponse(i, log[i][x],$
x, TRUE, $majoritiesPerIndex[x])$
$: x \in indices\}$

IN $\land$ $msgsReplicasSwitch' = msgsReplicasSwitch \cup msgsAsSet$

Update the commit index at the replica

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

$\land$ UNCHANGED $\langle leaderVars, switchVars, responsesToClient,$
$messages, msgsClientSwitch, log, currentTerm,$
$isActive, replicaSession, state\rangle$

Replica i receives an *AppendEntries* request from Replica j.
This just handles $m.entries$ of length 0 or 1, but
implementations could safely accept more by treating
them the same as multiple independent requests of 1 entry.

$HandleAppendEntriesRequest(i, j, m) \triangleq$

LET $logOk \triangleq$ $\lor$ $m.mprevLogIndex = 0$
$\lor$ $\land$ $m.mprevLogIndex > 0$
$\land$ $m.mprevLogIndex \leq Len(log[i])$
$\land$ $m.mprevLogTerm = log[i][m.mprevLogIndex].term$

IN $\land$ $m.mterm \leq currentTerm[i]$
$\land$ $\lor$ $\land$ reject request
$\lor$ $m.mterm < currentTerm[i]$
$\lor$ $\land$ $m.mterm = currentTerm[i]$
$\land$ $state[i] = Follower$
$\land$ $\neg logOk$
$\land$ $Send([mtype \mapsto AppendEntriesResponse,$
$mterm \mapsto currentTerm[i],$
$msuccess \mapsto$ FALSE,
$mmatchIndex \mapsto 0,$
$msource \mapsto i,$
$mdest \mapsto j])$
$\land$ UNCHANGED $\langle replicaVars\rangle$

process the request

$\lor$ $\land$ $m.mterm = currentTerm[i]$
$\land$ $state[i] = Follower$
$\land$ $logOk$
$\land$ LET $index \triangleq m.mprevLogIndex + 1$
IN $\lor$ already done with request
$\land$ $\lor$ $m.mentries = \langle\rangle$
$\lor$ $\land$ $Len(log[i]) \geq index$
$\land$ $m.mentries \neq \langle\rangle$
$\land$ $log[i][index].term = m.mentries[1].term$
This could make our *commitIndex* decrease (for

example if we process an old, duplicated request),
but that doesn't really affect anything.

$$
\begin{array}{l}
\quad\quad \land\ commitIndex' = [commitIndex\ \text{EXCEPT}\ ![i] = m.mcommitIndex] \\
\quad\quad \land\ Send([mtype \mapsto AppendEntriesResponse, \\
\quad\quad\quad\quad mterm \mapsto currentTerm[i], \\
\quad\quad\quad\quad msuccess \mapsto \text{TRUE}, \\
\quad\quad\quad\quad mmatchIndex \mapsto m.mprevLogIndex + Len(m.mentries), \\
\quad\quad\quad\quad msource \mapsto i, \\
\quad\quad\quad\quad mdest \mapsto j]) \\
\quad\quad \land\ \text{UNCHANGED}\ \langle log \rangle \\
\quad \lor\ \text{conflict: remove 1 entry} \\
\quad\quad \land\ m.mentries \neq \langle\rangle \\
\quad\quad \land\ Len(log[i]) \geq index \\
\quad\quad \land\ log[i][index].term \neq m.mentries[1].term \\
\quad\quad \land\ \text{LET}\ new \triangleq [index2 \in 1 \,..\, (Len(log[i]) - 1) \mapsto log[i][index2]] \\
\quad\quad\quad \text{IN}\ \ log' = [log\ \text{EXCEPT}\ ![i] = new] \\
\quad\quad \land\ Send([mtype \mapsto AppendEntriesResponse, \\
\quad\quad\quad\quad mterm \mapsto currentTerm[i], \\
\quad\quad\quad\quad msuccess \mapsto \text{FALSE}, \\
\quad\quad\quad\quad mmatchIndex \mapsto 0, \\
\quad\quad\quad\quad msource \mapsto i, \\
\quad\quad\quad\quad mdest \mapsto j]) \\
\quad\quad \land\ \text{UNCHANGED}\ \langle commitIndex \rangle \\
\quad \lor\ \text{no conflict: append entry} \\
\quad\quad \land\ m.mentries \neq \langle\rangle \\
\quad\quad \land\ Len(log[i]) = m.mprevLogIndex \\
\quad\quad \land\ log' = [log\ \text{EXCEPT}\ ![i] = Append(log[i],\ m.mentries[1])] \\
\quad\quad \land\ Send([mtype \mapsto AppendEntriesResponse, \\
\quad\quad\quad\quad mterm \mapsto currentTerm[i], \\
\quad\quad\quad\quad msuccess \mapsto \text{TRUE}, \\
\quad\quad\quad\quad mmatchIndex \mapsto m.mprevLogIndex + Len(m.mentries), \\
\quad\quad\quad\quad msource \mapsto i, \\
\quad\quad\quad\quad mdest \mapsto j]) \\
\quad\quad \land\ \text{UNCHANGED}\ \langle commitIndex \rangle \\
\land\ \text{UNCHANGED}\ \langle leaderVars,\ switchVars,\ responsesToClient, \\
\quad msgsClientSwitch,\ msgsReplicasSwitch, \\
\quad isActive,\ currentTerm,\ state,\ replicaSession \rangle
\end{array}
$$

Replica i receives an *AppendEntries* response from Replica j

$HandleAppendEntriesResponse(i, j, m) \triangleq$

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

$\land\ \lor\ \land\ m.msuccess$ successful

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

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

$\quad \lor\ \land\ \neg m.msuccess$ not successful

$\qquad \land\ nextIndex' = [nextIndex \text{ EXCEPT } ![i][j] = Max(\{nextIndex[i][j] - 1, 1\})]$

$\qquad \land\ \text{UNCHANGED } \langle matchIndex \rangle$

$\land\ Discard(m)$

$\land\ \text{UNCHANGED } \langle replicaVars, switchVars, responsesToClient, msgsClientSwitch, msgsReplicasSwitch, replicaKGroups \rangle$

process a message. The message will be processed
by its recipient based on its type

$Receive(m) \triangleq$

$\text{LET } i \triangleq m.mdest$

$\qquad j \triangleq m.msource$

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

$\land\ isActive[i] = ReplicaUpState$

$\land\ \lor\ \land\ m.mtype \in \{AppendEntriesRequest, AppendEntriesResponse\}$

$\qquad \land\ UpdateTerm(i, j, m)$

Append entry request from replica j to replica i

$\quad \lor\ \land\ m.mtype = AppendEntriesRequest$

$\qquad \land\ HandleAppendEntriesRequest(i, j, m)$

Append entry response from replica j to replica i

$\quad \lor\ \land\ m.mtype = AppendEntriesResponse$

$\qquad \land\ \lor\ DropStaleResponse(i, j, m)$

$\qquad \quad \lor\ HandleAppendEntriesResponse(i, j, m)$

Read request from the switch to replica i

$\quad \lor\ \land\ m.mtype = InternalReadRequest$

$\qquad \land\ ReplicaReceiveReadRequest(m, i)$

Write request from the switch to replica i

$\quad \lor\ \land\ m.mtype = InternalWriteRequest$

$\qquad \land\ ReplicaReceiveWriteRequest(m, i)$

$ReplicasReceiveRaftInternalMsgs \triangleq$

$\land\ \exists\, m \in messages : Receive(m)$

$ReplicasReceiveFromSwitch \triangleq$

$\land\ \exists\, m \in msgsReplicasSwitch :$

$\qquad \land\ m.mdest \neq SwitchIp \land Receive(m)$

Defines how all *variables*(*Replicas*, *Client*, *Switch*)
may transition.

$$
\begin{array}{l}
\mathit{Next} \triangleq \\
\quad \text{client transitions} \\
\quad \land \lor \exists \mathit{key} \in \mathit{KeySpace} : \mathit{IssueReadRequest}(\mathit{key}) \\
\quad\quad \lor \exists \mathit{key} \in \mathit{KeySpace} : \exists \mathit{value} \in \mathit{ValueSpace} : \\
\quad\quad\quad\quad\quad\quad \mathit{IssueWriteRequest}(\mathit{key}, \mathit{value}) \\
\\
\quad\quad \text{Switch transitions} \\
\quad\quad \lor \mathit{switchFails} \\
\quad\quad \lor \mathit{SwitchReceiveFromClient} \\
\quad\quad \lor \land \mathit{Cardinality}(\mathit{msgsReplicasSwitch}) > 0 \\
\quad\quad\quad \land \text{LET } \mathit{messagesToSwitch} \triangleq \{x \in \mathit{msgsReplicasSwitch} : x.\mathit{mdest} = \mathit{SwitchIp}\} \\
\quad\quad\quad\quad \text{IN} \quad \land \exists \mathit{msg} \in \mathit{messagesToSwitch} : \mathit{SwitchReceiveFromReplica}(\mathit{msg}) \\
\\
\quad\quad \text{Replica transitions} \\
\quad\quad \lor \exists i \in \mathit{Replicas} : \mathit{Stop}(i) \\
\quad\quad \lor \exists i \in \mathit{Replicas} : \mathit{Start}(i) \\
\quad\quad \lor \mathit{ElectLeader} \\
\quad\quad \lor \mathit{ReplicasReceiveRaftInternalMsgs} \\
\quad\quad \lor \mathit{ReplicasReceiveFromSwitch} \\
\\
\quad\quad \text{Leader transitions} \\
\quad\quad \lor \exists i \in \mathit{Replicas} : \mathit{LeaderActivateSwitch}(i) \\
\quad\quad \lor \exists i, j \in \mathit{Replicas} : \mathit{AppendEntries}(i, j) \\
\quad\quad \lor \exists i \in \mathit{Replicas} \quad : \mathit{AdvanceCommitIndex}(i)
\end{array}
$$

Safety Invariants

In the following statements, each response has the logs of all replicas at the time it was served by a replica (This is only for safety check and should not be the case for real implementations)

Invariant that defines that read responses that passes the switch to the client are linearizable. Check the logs of all replica to get the last committed *log* index that update the requested key. The returned value "*msg.mvalue*" should equal the value in that *log* index.

$$
\begin{array}{l}
\mathit{isForwardedToClientReadSafe}(\mathit{response}) \triangleq \\
\quad \text{LET } \mathit{msg} \triangleq \mathit{response}.\mathit{msg} \\
\quad\quad\quad \mathit{logEntriesForKey} \triangleq \mathit{entriesForKey}(\mathit{msg}.\mathit{mleaderId}, \mathit{msg}.\mathit{mkey})
\end{array}
$$

$$
\begin{array}{l}
\quad\quad committedEntries \triangleq \{j \in logEntriesForKey : \\
\quad\quad\quad\quad\quad\quad AgreeIndex(j,\ msg.mallLogs,\ msg.mleaderId) \\
\quad\quad\quad\quad\quad\quad \in Quorum \land j \leq msg.mlogIndex\} \\
\quad\quad lastCommittedIndex \triangleq indexOfLastEntry(committedEntries) \\
\quad\quad lastEntry \triangleq log[msg.mleaderId][lastCommittedIndex] \\
\quad \text{IN} \\
\quad\quad \lor \land msg.mstatus = \text{TRUE} \\
\quad\quad\quad \land lastCommittedIndex = msg.mlogIndex \\
\quad\quad\quad \land lastEntry.key = msg.mkey \\
\quad\quad\quad \land lastEntry.value = msg.mvalue
\end{array}
$$

Invariant that defines that write responses that
passes the switch to the client are committed
on majority of the replicas.

$$
\begin{array}{l}
isForwardedToClientWriteCorrect(response) \triangleq \\
\quad \text{LET } msg \triangleq response.msg \\
\quad\quad\quad isOnMajority \triangleq AgreeIndex(msg.mlogIndex, \\
\quad\quad\quad\quad\quad\quad\quad msg.mallLogs,\ msg.msource) \in Quorum \\
\quad \text{IN} \quad isOnMajority
\end{array}
$$

Invariant that defines the correctness
of all responses forwarded to the client

$$
\begin{array}{l}
InvResponsesToClientCorrectness \triangleq \\
\quad \forall\, res \in responsesToClient : \\
\quad\quad \lor \land res.msg.mtype = ReadResponse \\
\quad\quad\quad \land isForwardedToClientReadSafe(res) \\
\quad\quad \lor \land res.msg.mtype = WriteResponse \\
\quad\quad\quad \land isForwardedToClientWriteCorrect(res)
\end{array}
$$

Invariant that defines that no two entries have
the same sequence number or *log* index.

$$
\begin{array}{l}
InvSwitchRegisterCorrectness \triangleq \\
\quad \land \forall\, i,\ j \in 1 \mathrel{..} SwitchKGroupsNum : \\
\quad\quad\quad \lor \land i \neq j \\
\quad\quad\quad\quad \land switchKGourpArray[i].leaderAcked = \text{TRUE} \\
\quad\quad\quad\quad \land switchKGourpArray[j].leaderAcked = \text{TRUE} \\
\quad\quad\quad\quad \land switchKGourpArray[i].seqNum \neq switchKGourpArray[j].seqNum \\
\quad\quad\quad\quad \land switchKGourpArray[i].logIndex \neq switchKGourpArray[j].logIndex \\
\quad\quad\quad \lor i = j \\
\quad\quad\quad \lor switchKGourpArray[i].leaderAcked = \text{FALSE} \\
\quad\quad\quad \lor switchKGourpArray[j].leaderAcked = \text{FALSE}
\end{array}
$$

invariant that defines that no two leaders exist
at the same time

$InvLeaderElectionSafety \triangleq$

$$
\begin{array}{ll}
\text{LET} & \mathit{runningReplicas} \triangleq \{i \in \mathit{Replicas} : \mathit{isActive}[i] = \mathit{ReplicaUpState}\} \\
 & \mathit{leaders} \qquad\qquad \triangleq \{i \in \mathit{runningReplicas} : \mathit{state}[i] = \mathit{Leader}\} \\
\text{IN} & \mathit{Cardinality}(\mathit{leaders}) \leq 1
\end{array}
$$