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

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

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

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

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*

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

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

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

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

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

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

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.

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

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

$msource \mapsto i,$
$mdest \mapsto SwitchIp]$

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

$createKGroup(leaderAcked,\ replicasIds,\ seqNum,\ logIndex) \triangleq$
$[leaderAcked \mapsto leaderAcked,$
$replicasIds \mapsto replicasIds,$
$seqNum \mapsto seqNum,$
$logIndex \mapsto logIndex]$

$createLogEntry(i,\ msg) \triangleq$
$[term \mapsto currentTerm[i],$
$key \mapsto msg.mkey,$
$value \mapsto msg.mvalue,$
$hash \mapsto msg.mhash,$
$seqNum \mapsto msg.mkGroupSeqNum,$
$switchId \mapsto msg.msession]$

$createResHistoryEntry(msg,\ tag) \triangleq$
$[msg \mapsto msg,$
$switchKGroupEntry \mapsto switchKGourpArray[getIndexFromHash(msg.mhash)],$
$tag \mapsto tag]$

Variables initialization

$InitSwitchVars \triangleq$
Initially the switch is inactive and

```
    KGroupArray is stable
    ∧ switchKGourpArray = [i ∈ 1 .. SwitchKGroupsNum
                            ↦ createKGroup(TRUE, {}, Nil, Nil)]
    ∧ switchState       = SwitchStateInactive
    ∧ switchTermId      = 0
    ∧ switchLeaderId    = Nil
    ∧ switchSeqNum      = 0
    ∧ session           = 0

InitMsgsSets ≜
    ∧ msgsClientSwitch     = {}
    ∧ msgsReplicasSwitch   = {}
    ∧ messages             = {}


InitReplicaVars ≜
    ∧ currentTerm = [i ∈ Replicas ↦ 1]
    ∧ state       = [i ∈ Replicas ↦ Follower]
    ∧ isActive    = [i ∈ Replicas ↦ ReplicaUpState]
    ∧ replicaSession = [i ∈ Replicas ↦ 0]
    ∧ log         = [i ∈ Replicas ↦ ⟨⟩]
    ∧ commitIndex = [i ∈ Replicas ↦ 0]

InitLeaderVars ≜
    ∧ nextIndex     = [i ∈ Replicas ↦ [j ∈ Replicas ↦ 1]]
    ∧ matchIndex    = [i ∈ Replicas ↦ [j ∈ Replicas ↦ 0]]
    ∧ replicaKGroups = [i ∈ Replicas ↦
                          [j ∈ 1 .. SwitchKGroupsNum ↦ 0]]

Init ≜ ∧ InitReplicaVars
       ∧ InitLeaderVars
       ∧ InitSwitchVars
       ∧ InitMsgsSets
        Needed to prove safety
       ∧ responsesToClient = {}
```

Variables actions

Client actions

client sends a read request to read key k

```
IssueReadRequest(key) ≜
    ∧ LET request ≜ createClinetReadRequest(key)
      IN  msgsClientSwitch' = AddToSet(msgsClientSwitch, request)
    ∧ UNCHANGED ⟨messages, replicaVars, leaderVars,
                 switchVars, msgsReplicasSwitch, responsesToClient⟩
```

client issues a write request to update key k

$IssueWriteRequest(key,\ value) \triangleq$
$\quad \wedge$ LET $request \triangleq createClinetWriteRequest(key,\ value)$
$\quad\quad$ IN $msgsClientSwitch' = AddToSet(msgsClientSwitch,\ request)$
$\quad \wedge$ UNCHANGED $\langle messages,\ replicaVars,\ leaderVars,$
$\quad\quad\quad\quad\quad\quad switchVars,\ msgsReplicasSwitch,\ responsesToClient\rangle$

Switch actions

Switch state changes from Active to inactive

$switchFails \triangleq$
$\quad \wedge\ switchState = SwitchStateActive$
$\quad \wedge\ switchState' = SwitchStateInactive$
$\quad \wedge$ UNCHANGED $\langle replicaVars,\ leaderVars,\ msgsVars,\ responsesToClient,$
$\quad\quad\quad\quad\quad\quad switchSeqNum,\ session,\ switchKGourpArray,$
$\quad\quad\quad\quad\quad\quad switchLeaderId,\ switchTermId\rangle$

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.

$SwitchHandleCleintRead(msg) \triangleq$
$\quad$ LET $kGroup \triangleq switchKGourpArray[getIndexFromHash(msg.mhash)]$
$\quad\quad forwardTo \triangleq$ IF $kGroup.leaderAcked \wedge$
$\quad\quad\quad\quad\quad\quad\quad\quad kGroup.seqNum \neq Nil$
$\quad\quad\quad\quad\quad$ THEN $ChooseRandomly(\{x \in kGroup.replicasIds :$
$\quad\quad\quad\quad\quad\quad\quad\quad x \neq switchLeaderId\})$
$\quad\quad\quad\quad\quad$ ELSE IF $kGroup.leaderAcked \wedge$
$\quad\quad\quad\quad\quad\quad\quad\quad kGroup.seqNum = Nil$
$\quad\quad\quad\quad\quad$ THEN $switchLeaderId$
$\quad\quad\quad\quad\quad$ ELSE $switchLeaderId$
$\quad\quad internalMsg \triangleq createInternalReadRequest(msg,\ kGroup,\ forwardTo)$

$\quad$ IN
$\quad\quad \wedge\ msgsReplicasSwitch' = AddToSet(msgsReplicasSwitch,\ internalMsg)$
$\quad\quad \wedge$ UNCHANGED $\langle replicaVars,\ leaderVars,\ switchVars,$
$\quad\quad\quad\quad\quad\quad responsesToClient,\ messages,\ msgsClientSwitch\rangle$

Switch handles a write request from a client

$SwitchHandleCleintWrite(msg) \triangleq$
$\quad$ Mark the *KGroup* associated with the key as unstable
$\quad$ LET $kGroup \triangleq switchKGourpArray[getIndexFromHash(msg.mhash)]$

```
        updatedKGroup  ≜ [kGroup EXCEPT !.leaderAcked = FALSE,
                                         !.seqNum = switchSeqNum + 1,
                                         !.replicasIds = {},
                                         !.logIndex = Nil]
        internalMsg    ≜ createInternalWriteRequest(msg, updatedKGroup)

    IN
        ∧ msgsReplicasSwitch' = AddToSet(msgsReplicasSwitch, internalMsg)
        ∧ switchKGourpArray' = [switchKGourpArray EXCEPT
                                ![getIndexFromHash(msg.mhash)] = updatedKGroup]
        ∧ switchSeqNum' =    switchSeqNum + 1  Increments the sequence number
        ∧ UNCHANGED ⟨replicaVars, leaderVars, responsesToClient,
                     messages, msgsClientSwitch, switchState,
                     switchTermId, switchLeaderId, session⟩


 Switch receives a read or write request from a client
SwitchReceiveFromClient ≜
    ∧ switchState = SwitchStateActive
    ∧ Cardinality(msgsClientSwitch) > 0
    ∧ LET msg  ≜  ChooseRandomly(msgsClientSwitch)
          type ≜  msg.mtype
      IN  ∨ ∧ type = ClientReadRequest
            ∧ SwitchHandleCleintRead(msg)
          ∨ ∧ type = ClientWriteRequest
            ∧ SwitchHandleCleintWrite(msg)


 Switch handles a read response
SwitchHandleReadResponse(msg) ≜
    ∧ ∨ ∧ msg.msource ≠ switchLeaderId  msg is from follower
        ∧ msg.mterm = switchTermId   msg.term = switch term
        ∧ msg.mstatus = TRUE    The operation was succeeded
        Map the key to a KGroup based on hash.
        The response will be sent to teh client if
        1 − msg.seqNum = KGroup.seqNum
        2 − KGroup is stable
        otherwise the request will be dropped
        ∧ LET KGroup ≜ switchKGourpArray[getIndexFromHash(msg.mhash)]
              isSeqOk ≜ KGroup.seqNum = msg.mkGroupSeqNum

          IN  ∨ ∧ isSeqOk
                ∧ KGroup.leaderAcked
                ∧ responsesToClient' = AddToSet(
                                       responsesToClient,
                                       createResHistoryEntry(msg, Nil))
```

```
                    ∨ ∧ ¬isSeqOk
                      ∧ UNCHANGED ⟨responsesToClient⟩
                    ∨ ∧ ¬KGroup.leaderAcked
                      ∧ isSeqOk
                      ∧ UNCHANGED ⟨responsesToClient⟩
            ∨ ∧ msg.msource ≠ switchLeaderId
              ∧ msg.mstatus = FALSE  The key does not exist
              ∧ UNCHANGED ⟨responsesToClient⟩
             The operation succeeded and the response
             is from the current leader. Just forward it to the client
            ∨ ∧ msg.msource = switchLeaderId
              ∧ msg.mterm = switchTermId
              ∧ msg.mstatus = TRUE
              ∧ responsesToClient' = AddToSet(responsesToClient,
                                              createResHistoryEntry(msg, Nil))
            ∨ ∧ msg.msource = switchLeaderId
              ∧ msg.mstatus = FALSE  The key does not exist
              ∧ UNCHANGED ⟨responsesToClient⟩
        ∧ UNCHANGED ⟨replicaVars, leaderVars, switchVars, msgsVars⟩

 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) ≜
    ∧ ∨ ∧ msg.msource = switchLeaderId  Message from the leader
      ∧ msg.mterm   = switchTermId  Message.term = switch.term
      ∧ msg.mstatus = TRUE  The operation succeeded
      ∧ LET KGroup ≜ switchKGourpArray[getIndexFromHash(msg.mhash)]
            isSeqOk ≜ KGroup.seqNum = msg.mkGroupSeqNum
            updatedKGroup ≜ [KGroup EXCEPT
                                ! .leaderAcked = TRUE,
                                ! .replicasIds = msg.mreplicaIds,
                                ! .logIndex    = msg.mlogIndex]
             msg.seqNum = KGroup.seqNum
        IN   ∨ ∧ isSeqOk
              Mark the KGroup as stable
             ∧ switchKGourpArray' = [switchKGourpArray EXCEPT
                                        ![getIndexFromHash(msg.mhash)] =
                                            updatedKGroup]
             ∧ responsesToClient' = AddToSet(responsesToClient,
                                             createResHistoryEntry(msg, Nil))
```

$msg.seqNum \,!=\, KGroup.seqNum$

$\lor \land \neg isSeqOk$
$\land responsesToClient' = AddToSet(responsesToClient,$
$createResHistoryEntry(msg,\ Nil))$
$\land$ UNCHANGED $\langle switchKGourpArray \rangle$

$\lor \land msg.msource \neq switchLeaderId$
$\land$ UNCHANGED $\langle switchKGourpArray,\ responsesToClient \rangle$

$\lor \land msg.mstatus = \text{FALSE}$
$\land$ UNCHANGED $\langle switchKGourpArray,\ responsesToClient \rangle$

$\land$ UNCHANGED $\langle replicaVars,\ leaderVars,\ msgsVars,$
$switchSeqNum,\ session,\ switchLeaderId,$
$switchTermId,\ switchState \rangle$

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

$\lor \land switchState = SwitchStateActive$
$\land msg.msession = session$
$\land msg.mterm > switchTermId$
$\land switchState' = SwitchStateInactive$
$\land$ UNCHANGED $\langle replicaVars,\ leaderVars,\ msgsVars,$
$msgsClientSwitch,\ switchVars,\ responsesToClient \rangle$

msg is coming from an old leader
$\Rightarrow$ switch just ignore the message

$\lor \land switchState = SwitchStateActive$
$\land msg.msession = session$
$\land msg.mterm < switchTermId$
$\land$ UNCHANGED $\langle replicaVars,\ leaderVars,\ msgsVars,$
$responsesToClient,\ switchVars \rangle$

$msg.switchId$ does not match $switchId$
$\Rightarrow$ switch just ignore the message

$\lor \land switchState = SwitchStateActive$
$\land msg.msession \neq session$
$\land$ UNCHANGED $\langle replicaVars,\ leaderVars,\ msgsVars,$
$responsesToClient,\ switchVars \rangle$

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

$\lor \land switchState = SwitchStateActive$
$\land msg.msession = session$
$\land msg.mtype = ReadResponse$
$\land SwitchHandleReadResponse(msg)$

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

$\lor \land switchState = SwitchStateActive$
$\quad \land msg.msession = session$
$\quad \land msg.mtype = WriteResponse$
$\quad \land SwitchHandleWriteResponse(msg)$

Replica actions

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

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

Replica i becomes active. The replica stars as follower

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

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.

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

```
        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 ≜ [j ∈ startIndex .. endIndex ↦
                       j ≤ commitIndex[i]]
    Get the set of replicas that
    acknowleged each log entry
  keysToReplicas  ≜ [j ∈ startIndex .. endIndex ↦
                       IF leaderAckedFlag[j]
                        THEN AgreeIndex(j, log, i)
                        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
 ∧ switchKGourpArray' =
    [j ∈ 1 .. SwitchKGroupsNum ↦
    IF   mapLogEntryToKGroupIndex[j] ≠ Nil
     THEN createKGroup(leaderAckedFlag[mapLogEntryToKGroupIndex[j]],
                       keysToReplicas[mapLogEntryToKGroupIndex[j]],
                       0, mapLogEntryToKGroupIndex[j])
     ELSE createKGroup(TRUE, Replicas, Nil, Nil)]


 Leader i activates the switch
LeaderActivateSwitch(i) ≜
  ∧ state[i] = Leader  Replica is the leader
  ∧ isActive[i]  = ReplicaUpState  Replica is active
  ∧ switchState = SwitchStateInactive  Switch is inactive
  ∧ replicaSession' = [replicaSession EXCEPT ![i] = session + 1]
  ∧ session' = session + 1  Update the replica and switch sessions
  ∧ switchTermId' = currentTerm[i]  Update switch term number
  ∧ switchLeaderId' = i  Update leader id of the switch
  ∧ switchSeqNum' = 0  Reset the sequence number
  ∧ fillSwitchKGroup(i)  Populate switch KGroup array
  ∧ switchState' = SwitchStateActive  activate the switch
  ∧ UNCHANGED ⟨leaderVars, msgsVars, responsesToClient,
               state, log, commitIndex, currentTerm,
               isActive⟩


 Any RPC with a newer term causes the recipient to advance its term first.
UpdateTerm(i, j, m) ≜
  ∧ m.mterm > currentTerm[i]
  ∧ currentTerm'   = [currentTerm EXCEPT ![i] = m.mterm]
  ∧ state'         = [state       EXCEPT ![i]  = Follower]
    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.
$DropStaleResponse(i,\ j,\ m) \triangleq$
$\land\ m.mterm < currentTerm[i]$
$\land$ 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,
```

```
                    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], lastEntry}),
                    msource         ↦ i,
                    mdest           ↦ j]
  IN    ∧ Len(entries) > 0
        ∧ Send(m)
∧ UNCHANGED ⟨replicaVars, leaderVars, switchVars,
             msgsClientSwitch, msgsReplicasSwitch,
             responsesToClient⟩
```

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

```
AdvanceCommitIndex(i) ≜
  ∧ state[i] = Leader
  ∧ isActive[i] = ReplicaUpState
  ∧ LET The set of replicas that agree up through an index.
        Agree(index) ≜ {i} ∪ {k ∈ Replicas :
                               matchIndex[i][k] ≥ index}
        The maximum indexes for which a quorum agrees
        agreeIndexes ≜ {index ∈ 1 .. Len(log[i]) :
                         Agree(index) ∈ Quorum}
        Entries that are replicated to majorities
        but not yet committed
        IndicesToCommit ≜ {index ∈ agreeIndexes : index > commitIndex[i]}
        majoritiesPerIndex ≜ [index ∈ agreeIndexes ↦ Agree(index)]
        New value for commitIndex'[i]
        newCommitIndex ≜
          IF ∧ agreeIndexes ≠ {}
             ∧ log[i][Max(agreeIndexes)].term = currentTerm[i]
           THEN Max(agreeIndexes)
           ELSE commitIndex[i]
    IN  ∧ newCommitIndex > commitIndex[i]
        For each entry that will be committed, send a write
        response to clients
        ∧ LET indices ≜ commitIndex[i] + 1 .. newCommitIndex
              msgs ≜ [x ∈ indices ↦
                        createWriteResponse(i, log[i][x],
```

```
                                                    x, TRUE, majoritiesPerIndex[x])]
            msgsAsSet ≜ {createWriteResponse(i, log[i][x],
                                                    x, TRUE, majoritiesPerIndex[x])
                                                      : x ∈ indices}

    IN   ∧ msgsReplicasSwitch' = msgsReplicasSwitch ∪ msgsAsSet
         Update the commit index at the replica
         ∧ commitIndex' = [commitIndex EXCEPT ![i] = newCommitIndex]
         ∧ UNCHANGED ⟨leaderVars, switchVars, responsesToClient,
                      messages, msgsClientSwitch, log, currentTerm,
                      isActive, replicaSession, state⟩

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) ≜
  LET logOk ≜ ∨ m.mprevLogIndex = 0
              ∨ ∧ m.mprevLogIndex > 0
                ∧ m.mprevLogIndex ≤ Len(log[i])
                ∧ m.mprevLogTerm = log[i][m.mprevLogIndex].term
  IN ∧ m.mterm ≤ currentTerm[i]
     ∧ ∨ ∧ reject request
         ∨ m.mterm < currentTerm[i]
         ∨ ∧ m.mterm = currentTerm[i]
           ∧ state[i] = Follower
           ∧ ¬logOk
       ∧ Send([mtype      ↦ AppendEntriesResponse,
               mterm      ↦ currentTerm[i],
               msuccess   ↦ FALSE,
               mmatchIndex ↦ 0,
               msource    ↦ i,
               mdest      ↦ j])
       ∧ UNCHANGED ⟨replicaVars⟩
       process the request
     ∨ ∧ m.mterm = currentTerm[i]
       ∧ state[i] = Follower
       ∧ logOk
       ∧ LET index ≜ m.mprevLogIndex + 1
         IN ∨ already done with request
              ∧ ∨ m.mentries = ⟨⟩
                ∨ ∧ Len(log[i]) ≥ index
                  ∧ m.mentries ≠ ⟨⟩
                  ∧ log[i][index].term = m.mentries[1].term
                This could make our commitIndex decrease (for
```

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

$\land\ commitIndex' = [commitIndex \text{ EXCEPT } ![i] = m.mcommitIndex]$
$\land\ Send([mtype \mapsto AppendEntriesResponse,$
$\quad mterm \mapsto currentTerm[i],$
$\quad msuccess \mapsto \text{TRUE},$
$\quad mmatchIndex \mapsto m.mprevLogIndex + Len(m.mentries),$
$\quad msource \mapsto i,$
$\quad mdest \mapsto j])$
$\land\ \text{UNCHANGED } \langle log \rangle$

$\lor$ conflict: remove 1 entry
$\land\ m.mentries \neq \langle\rangle$
$\land\ Len(log[i]) \geq index$
$\land\ log[i][index].term \neq m.mentries[1].term$
$\land\ \text{LET } new \triangleq [index2 \in 1\,..\,(Len(log[i]) - 1) \mapsto log[i][index2]]$
$\quad \text{IN } log' = [log \text{ EXCEPT } ![i] = new]$

$\land\ Send([mtype \mapsto AppendEntriesResponse,$
$\quad mterm \mapsto currentTerm[i],$
$\quad msuccess \mapsto \text{FALSE},$
$\quad mmatchIndex \mapsto 0,$
$\quad msource \mapsto i,$
$\quad mdest \mapsto j])$
$\land\ \text{UNCHANGED } \langle commitIndex \rangle$

$\lor$ no conflict: append entry
$\land\ m.mentries \neq \langle\rangle$
$\land\ Len(log[i]) = m.mprevLogIndex$
$\land\ log' = [log \text{ EXCEPT } ![i] = Append(log[i],\ m.mentries[1])]$
$\land\ Send([mtype \mapsto AppendEntriesResponse,$
$\quad mterm \mapsto currentTerm[i],$
$\quad msuccess \mapsto \text{TRUE},$
$\quad mmatchIndex \mapsto m.mprevLogIndex + Len(m.mentries),$
$\quad msource \mapsto i,$
$\quad mdest \mapsto j])$
$\land\ \text{UNCHANGED } \langle commitIndex \rangle$

$\land\ \text{UNCHANGED } \langle leaderVars,\ switchVars,\ responsesToClient,$
$msgsClientSwitch,\ msgsReplicasSwitch,$
$isActive,\ currentTerm,\ state,\ replicaSession \rangle$

Replica i receives an *AppendEntries* response from Replica j

```
HandleAppendEntriesResponse(i, j, m) ≜
    ∧ m.mterm = currentTerm[i]
    ∧ ∨ ∧ m.msuccess  successful
        ∧ nextIndex'  = [nextIndex  EXCEPT ![i][j]  = m.mmatchIndex + 1]
        ∧ matchIndex' = [matchIndex EXCEPT ![i][j] = m.mmatchIndex]
      ∨ ∧ ¬m.msuccess  not successful
        ∧ nextIndex' = [nextIndex EXCEPT ![i][j] =
                            Max({nextIndex[i][j] − 1, 1})]
        ∧ UNCHANGED ⟨matchIndex⟩
    ∧ Discard(m)
    ∧ UNCHANGED ⟨replicaVars, switchVars, responsesToClient,
                 msgsClientSwitch, msgsReplicasSwitch, replicaKGroups⟩


 process a message. The message will be processed
 by its recipient based on its type
Receive(m) ≜
    LET i ≜ m.mdest
        j ≜ m.msource
    IN   Any RPC with a newer term causes the recipient to advance
         its term first. Responses with stale terms are ignored.
    ∧   isActive[i] = ReplicaUpState
    ∧   ∨ ∧ m.mtype ∈ {AppendEntriesRequest, AppendEntriesResponse}
          ∧ UpdateTerm(i, j, m)
         Append entry request from replica j to replica i
        ∨ ∧ m.mtype = AppendEntriesRequest
          ∧ HandleAppendEntriesRequest(i, j, m)
         Append entry response from replica j to replica i
        ∨ ∧ m.mtype = AppendEntriesResponse
          ∧ ∨ DropStaleResponse(i, j, m)
            ∨ HandleAppendEntriesResponse(i, j, m)

         Read request from the switch to replica i
        ∨ ∧ m.mtype = InternalReadRequest
          ∧ ReplicaReceiveReadRequest(m, i)
         Write request from the switch to replica i
        ∨ ∧ m.mtype = InternalWriteRequest
          ∧ ReplicaReceiveWriteRequest(m, i)

ReplicasReceiveRaftInternalMsgs ≜
    ∧ ∃ m ∈ messages : Receive(m)

ReplicasReceiveFromSwitch ≜
    ∧ ∃ m ∈ msgsReplicasSwitch :
            ∧ m.mdest ≠ SwitchIp ∧ Receive(m)
```

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

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

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.

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

```
        committedEntries ≜ {j ∈ logEntriesForKey :
                            AgreeIndex(j, msg.mallLogs, msg.mleaderId)
                             ∈ Quorum ∧ j ≤ msg.mlogIndex}
        lastCommittedIndex ≜ indexOfLastEntry(committedEntries)
        lastEntry ≜ log[msg.mleaderId][lastCommittedIndex]
    IN
        ∨ ∧ msg.mstatus = TRUE
          ∧ lastCommittedIndex = msg.mlogIndex
          ∧ lastEntry.key      = msg.mkey
          ∧ lastEntry.value    = msg.mvalue

 Invariant that defines that write responses that
 passes the switch to the client are committed
 on majority of the replicas.
isForwardedToClientWriteCorrect(response) ≜
    LET msg ≜ response.msg
        isOnMajority ≜ AgreeIndex(msg.mlogIndex,
                                  msg.mallLogs, msg.msource) ∈ Quorum
    IN  isOnMajority

 Invariant that defines the correctness
 of all responses forwarded to the client
InvResponsesToClientCorrectness ≜
    ∀ res ∈ responsesToClient :
      ∨ ∧ res.msg.mtype = ReadResponse
        ∧ isForwardedToClientReadSafe(res)
      ∨ ∧ res.msg.mtype = WriteResponse
        ∧ isForwardedToClientWriteCorrect(res)

 Invariant that defines that no two entries have
 the same sequence number or log index.
InvSwitchRegisterCorrectness ≜
    ∧ ∀ i, j ∈ 1 .. SwitchKGroupsNum :
        ∨ ∧ i ≠ j
          ∧ switchKGourpArray[i].leaderAcked = TRUE
          ∧ switchKGourpArray[j].leaderAcked = TRUE
          ∧ switchKGourpArray[i].seqNum ≠ switchKGourpArray[j].seqNum
          ∧ switchKGourpArray[i].logIndex ≠ switchKGourpArray[j].logIndex
        ∨ i = j
        ∨ switchKGourpArray[i].leaderAcked = FALSE
        ∨ switchKGourpArray[j].leaderAcked = FALSE


 invariant that defines that no two leaders exist
 at the same time
InvLeaderElectionSafety ≜
```

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