MODULE *Controller*

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *Messages*

The set of all *ONOS* nodes
CONSTANTS *Nodes*

The following variables are used by the *mastership* election service. These variables represent global atomic state.

The current *mastership* term
VARIABLE *term*

The current master node *ID*
VARIABLE *master*

A sequence of standby nodes
VARIABLE *backups*

The following variables are per-node variables used by controller nodes in the *mastership* arbitration protocol.

A queue of events from the *mastership* service to the node
VARIABLE *events*

The current term, master, and backups known to the node
VARIABLE *mastership*

The highest term sent to the device by the node
VARIABLE *sentTerm*

Whether the node has received a *MasterArbitrationUpdate* indicating it is the master
VARIABLE *isMaster*

The following variables are used to enforce state constraints during model checking.

A counter used to generate unique stream *IDs*
VARIABLE *streamId*

Stream change counter used for enforcing state constraints
VARIABLE *streamChanges*

Mastership change count used for enforcing state constraints
VARIABLE *mastershipChanges*

A count of all attempted writes to the switch
VARIABLE *writeCount*

Mastership/consensus related variables

$mastershipVars \triangleq \langle term,\ master,\ backups,\ mastershipChanges \rangle$

Mastership arbitration variables

$arbitrationVars \triangleq \langle streamVars,\ streamId,\ streamChanges \rangle$

Mastership event variables

$eventVars \triangleq \langle events,\ mastership \rangle$

Node related variables

$nodeVars \triangleq \langle events,\ mastership,\ sentTerm,\ streamId,\ streamChanges,\ isMaster,\ writeCount \rangle$

This section models the *mastership* election service used by the controller to elect masters. Mastership changes through join and leave steps. Mastership is done through a consensus service, so these steps are atomic. When a node joins or leaves the *mastership* election, events are queued to notify nodes of the *mastership* change. Nodes learn of *mastership* changes independently of the state change in the consensus service.

Returns the set of values in f

$Range(f) \triangleq \{f[x] : x \in \text{DOMAIN } f\}$

Returns a sequences with the element at the given index removed

$Drop(q,\ i) \triangleq SubSeq(q,\ 1,\ i-1) \circ SubSeq(q,\ i+1,\ Len(q))$

Node 'n' joins the *mastership* election

If the current 'master' is *Nil*, set the master to node 'n', increment the 'term', and send a *mastership* change event to each node. If the current 'master' is non-Nil, append node 'n' to the sequence of 'backups'.

$$
\begin{array}{l}
JoinMastershipElection(n) \triangleq \\
\quad \land \lor \land master = Nil \\
\qquad \land term' = term + 1 \\
\qquad \land master' = n \\
\qquad \land backups' = \langle\rangle \\
\qquad \land events' = [i \in Nodes \mapsto Append(events[i], [\\
\qquad\qquad\qquad\qquad\qquad term \mapsto term', \\
\qquad\qquad\qquad\qquad\qquad master \mapsto master', \\
\qquad\qquad\qquad\qquad\qquad backups \mapsto backups'])] \\
\qquad \lor \land master \neq Nil \\
\qquad\quad \land master \neq n \\
\qquad\quad \land n \notin Range(backups) \\
\qquad\quad \land backups' = Append(backups,\ n) \\
\qquad\quad \land events' = [i \in Nodes \mapsto Append(events[i], [\\
\qquad\qquad\qquad\qquad\qquad term \mapsto term, \\
\qquad\qquad\qquad\qquad\qquad master \mapsto master, \\
\qquad\qquad\qquad\qquad\qquad backups \mapsto backups'])]
\end{array}
$$

```
        ∧ UNCHANGED ⟨term, master⟩
    ∧ mastershipChanges′ = mastershipChanges + 1
    ∧ UNCHANGED ⟨mastership, sentTerm, isMaster, writeCount, messageVars, arbitrationVars⟩
```

Node 'n' leaves the *mastership* election

If node 'n' is the current 'master' and a backup exists, increment the 'term', promote the first backup to master, and send a *mastership* change event to each node. If node 'n' is the current 'master' and no backups exist, set the 'master' to *Nil*. If node 'n' is in the sequence of 'backups', simply remove it.

```
LeaveMastershipElection(n) ≜
    ∧ ∨ ∧ master = n
        ∧ ∨ ∧ Len(backups) > 0
            ∧ term′ = term + 1
            ∧ master′ = backups[1]
            ∧ backups′ = Pop(backups)
            ∧ events′ = [i ∈ Nodes ↦ Append(events[i], [
                                        term ↦ term′,
                                        master ↦ master′,
                                        backups ↦ backups′])]
          ∨ ∧ Len(backups) = 0
            ∧ master′ = Nil
            ∧ UNCHANGED ⟨term, backups, events⟩
      ∨ ∧ n ∈ Range(backups)
        ∧ backups′ = Drop(backups, CHOOSE j ∈ DOMAIN backups : backups[j] = n)
        ∧ UNCHANGED ⟨term, master, events⟩
    ∧ mastershipChanges′ = mastershipChanges + 1
    ∧ UNCHANGED ⟨mastership, sentTerm, isMaster, writeCount, messageVars, arbitrationVars⟩
```

This section models controller-side stream management.

Opens a new stream on the controller side

```
OpenStream(n) ≜
    ∧ requestStream[n].state = Closed
    ∧ streamId′ = streamId + 1
    ∧ requestStream′ = [requestStream EXCEPT ![n] = [id ↦ streamId′, state ↦ Open]]
    ∧ requests′ = [requests EXCEPT ![n] = ⟨⟩]
    ∧ responses′ = [responses EXCEPT ![n] = ⟨⟩]
    ∧ streamChanges′ = streamChanges + 1
    ∧ UNCHANGED ⟨mastershipVars, eventVars, sentTerm, isMaster, responseStream, writeCount⟩
```

Closes an open stream on the controller side

```
CloseStream(n) ≜
    ∧ requestStream[n].state = Open
    ∧ requestStream′ = [requestStream EXCEPT ![n].state = Closed]
    ∧ sentTerm′ = [sentTerm EXCEPT ![n] = 0]
```

$\land\ isMaster' = [isMaster \text{ EXCEPT } ![n] = \text{FALSE}]$
$\land\ streamChanges' = streamChanges + 1$
$\land\ \text{UNCHANGED } \langle mastershipVars,\ eventVars,\ responseStream,\ messageVars,\ streamId,\ writeCount\rangle$

This section models controller-side *mastership* arbitration. The controller nodes receive *mastership* change events from the *mastership* service and send master arbitration requests to the device. Additionally, master nodes can send write requests to the device.

Returns master node 'n' *election_id* for *mastership* term 'm'

$MasterElectionId(m) \triangleq m.term + Cardinality(Nodes)$

Returns the index of node 'n' in the sequence of 'm' backups

$BackupIndex(n,\ m) \triangleq \text{CHOOSE } i \in \text{DOMAIN } m.backups : m.backups[i] = n$

Returns backup node 'n' *election_id* for *mastership* term 'm'

$BackupElectionId(n,\ m) \triangleq MasterElectionId(m) - BackupIndex(n,\ m)$

Returns the *mastership* term for *MasterArbitrationUpdate* 'm'

$MasterTerm(m) \triangleq m.election\_id - Cardinality(Nodes)$

Node 'n' receives a *mastership* change event from the *mastership* service

When a *mastership* change event is received, the node's local *mastership* state is updated. If the *mastership* term has changed, the node will set a flag to push the *mastership* change to the device in the master arbitration step.

$LearnMastership(n) \triangleq$
$\land\ Len(events[n]) > 0$
$\land\ \text{LET } e \triangleq events[n][1]$
$\qquad m \triangleq mastership[n]$
$\quad \text{IN}$
$\qquad \lor\ \land\ e.term > m.term$
$\qquad\quad \land\ mastership' = [mastership \text{ EXCEPT } ![n] = [$
$\qquad\qquad\qquad\qquad term \mapsto e.term,$
$\qquad\qquad\qquad\qquad master \mapsto e.master,$
$\qquad\qquad\qquad\qquad backups \mapsto e.backups]]$
$\qquad \lor\ \land\ e.term = m.term$
$\qquad\quad \land\ mastership' = [mastership \text{ EXCEPT } ![n] = [$
$\qquad\qquad\qquad\qquad term \mapsto e.term,$
$\qquad\qquad\qquad\qquad master \mapsto e.master,$
$\qquad\qquad\qquad\qquad backups \mapsto e.backups]]$
$\land\ events' = [events \text{ EXCEPT } ![n] = Pop(events[n])]$
$\land\ \text{UNCHANGED } \langle mastershipVars,\ sentTerm,\ isMaster,\ writeCount,\ messageVars,\ arbitrationVars\rangle$

Node 'n' sends a *MasterArbitrationUpdate* to the device

If the node has an open stream to the device and a valid *mastership* state, a *MasterArbitrationUpdate* is sent to the device. If the node is a backup, the request's 'election_id' is set to (*mastership* term) + (number of nodes) − (backup index). If the node is the master, the 'election_id' is set to (*mastership* term) + (number of nodes). This is done to avoid $election\_ids \leq 0$. Note that the actual protocol requires a (*device_id*, *role_id*, *election_id*) tuple, but (*device_id*, *role_id*) have been excluded from this model as we're modelling interaction only within a single (*device_id*, *role_id*) and thus they're irrelevant to correctness. The *mastership* term is sent in *MasterArbitrationUpdate* requests for model checking.

$$
\begin{array}{l}
SendMasterArbitrationUpdate(n) \triangleq \\
\quad \land\ requestStream[n].state = Open \\
\quad \land\ \text{LET}\ m \triangleq mastership[n] \\
\quad\ \ \text{IN} \\
\qquad \land\ m.term > 0 \\
\qquad \land\ sentTerm[n] < m.term \\
\qquad \land\ \lor\ \land\ m.master = n \\
\qquad\qquad \land\ SendRequest(n, [\\
\qquad\qquad\qquad type \mapsto MasterArbitrationUpdate, \\
\qquad\qquad\qquad election\_id \mapsto MasterElectionId(m), \\
\qquad\qquad\qquad epoch \mapsto m.term]) \\
\qquad\qquad\ \lor\ \land\ m.master \neq n \\
\qquad\qquad\ \ \land\ n \in Range(m.backups) \\
\qquad\qquad\ \ \land\ SendRequest(n, [\\
\qquad\qquad\qquad type \mapsto MasterArbitrationUpdate, \\
\qquad\qquad\qquad election\_id \mapsto BackupElectionId(n, m), \\
\qquad\qquad\qquad epoch \mapsto m.term]) \\
\qquad \land\ sentTerm' = [sentTerm\ \text{EXCEPT}\ ![n] = m.term] \\
\quad \land\ \text{UNCHANGED}\ \langle mastershipVars, eventVars, isMaster, writeCount, arbitrationVars, responses \rangle
\end{array}
$$

Node 'n' receives a *MasterArbitrationUpdate* from the device

If the node has an open stream with a *MasterArbitrationUpdate*, determine whether the local node is the master. If the *MasterArbitrationUpdate* 'status' is *Ok*, the 'election_id' matches the last requested *mastership* term, and 'n' is the master for that term, update the node's state to master. Otherwise, the *mastership* request is considered out of date.

Note that the separate 'isMaster' state is maintained to indicate whether the *device* considers this node to be the current master, and this is necessary for the safety of the algorithm. Both the node and the device must agree on the role of the node.

$$
\begin{array}{l}
ReceiveMasterArbitrationUpdate(n) \triangleq \\
\quad \land\ requestStream[n].state = Open \\
\quad \land\ HasResponse(n, MasterArbitrationUpdate) \\
\quad \land\ \text{LET}\ r \triangleq NextResponse(n) \\
\qquad\ \ m \triangleq mastership[n] \\
\quad\ \ \text{IN} \\
\qquad \land\ \lor\ \land\ r.status\ \ = Ok \\
\qquad\qquad \land\ m.master = n \\
\qquad\qquad \land\ m.term = MasterTerm(r) \\
\qquad\qquad \land\ sentTerm[n] = m.term \\
\qquad\qquad \land\ isMaster' = [isMaster\ \text{EXCEPT}\ ![n] = \text{TRUE}]
\end{array}
$$

```
            ∨ ∧ ∨ r.status    ≠ Ok
              ∨ m.master ≠ n
              ∨ sentTerm[n] ≠ m.term
              ∨ m.term ≠ MasterTerm(r)
            ∧ isMaster′ = [isMaster EXCEPT ![n] = FALSE]
    ∧ DiscardResponse(n)
    ∧ UNCHANGED ⟨mastershipVars, eventVars, sentTerm, arbitrationVars, requests, writeCount⟩
```

Master node 'n' sends a *WriteRequest* to the device

To write to the device, the node must have an open stream, must have received a *mastership* change event from the *mastership* service (stored in 'mastership') indicating it is the master, and must have received a *MasterArbitrationUpdate* from the switch indicating it is the master (stored in 'isMaster') for the same term as was indicated by the *mastership* service. The term is sent with the *WriteRequest* for model checking.

```
SendWriteRequest(n) ≜
    ∧ requestStream[n].state = Open
    ∧ LET m ≜ mastership[n]
      IN
          ∧ m.term > 0
          ∧ m.master = n
          ∧ isMaster[n]
          ∧ writeCount′ = writeCount + 1
          ∧ SendRequest(n, [
                type        ↦ WriteRequest,
                election_id ↦ MasterElectionId(m),
                term        ↦ m.term])
    ∧ UNCHANGED ⟨mastershipVars, eventVars, arbitrationVars, isMaster, sentTerm, responses⟩
```

Node 'n' receives a write response from the device

```
ReceiveWriteResponse(n) ≜
    ∧ requestStream[n].state = Open
    ∧ HasResponse(n, WriteResponse)
    ∧ LET m ≜ NextResponse(n)
      IN
          ∨ m.status = Ok
          ∨ m.status = PermissionDenied
    ∧ DiscardResponse(n)
    ∧ UNCHANGED ⟨mastershipVars, nodeVars, arbitrationVars, requests⟩
```

```
\* Modification History
\* Last modified Thu Feb 21 16:26:11 PST 2019 by jordanhalterman
\* Created Wed Feb 20 23:49:08 PST 2019 by jordanhalterman
```