MODULE *Controller*

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

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

The current state of mastership elections
VARIABLES *term*, *master*, *backups*

The current mastership event queue for each node
VARIABLE *events*

The current mastership state for each node
VARIABLE *masterships*

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

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

Mastership/consensus related variables
$mastershipVars \triangleq \langle term,\ master,\ backups,\ mastershipChanges \rangle$

Node related variables
$nodeVars \triangleq \langle events,\ masterships,\ isMaster \rangle$

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'.

$$
\begin{array}{l}
JoinMastershipElection(n) \triangleq \\
\quad \land \lor \land master = Nil \\
\qquad \land term' = term + 1
\end{array}
$$

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

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.

$$
\begin{array}{l}
LeaveMastershipElection(n) \triangleq \\
\land\ \lor\ \land\ master = n \\
\qquad \land\ \lor\ \land\ Len(backups) > 0 \\
\qquad\qquad \land\ term' = term + 1 \\
\qquad\qquad \land\ master' = backups[1] \\
\qquad\qquad \land\ backups' = Pop(backups) \\
\qquad\qquad \land\ events' = [i \quad \in Nodes \mapsto Append(events[i], [\\
\qquad\qquad\qquad\qquad\qquad\qquad term \mapsto term', \\
\qquad\qquad\qquad\qquad\qquad\qquad master \mapsto master', \\
\qquad\qquad\qquad\qquad\qquad\qquad backups \mapsto backups'])] \\
\qquad\quad \lor\ \land\ Len(backups) = 0 \\
\qquad\qquad \land\ master' = Nil \\
\qquad\qquad \land\ \text{UNCHANGED}\ \langle term,\ backups,\ events\rangle \\
\quad \lor\ \land\ n \in Range(backups) \\
\qquad \land\ backups' = Drop(backups,\ \text{CHOOSE}\ j \in \text{DOMAIN}\ backups : backups[j] = n) \\
\qquad \land\ \text{UNCHANGED}\ \langle term,\ master,\ events\rangle \\
\land\ mastershipChanges' = mastershipChanges + 1 \\
\land\ \text{UNCHANGED}\ \langle masterships,\ isMaster,\ streamVars,\ messageVars\rangle
\end{array}
$$

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

$$BackupElectionId(n,\ m) \triangleq m.term + Cardinality(Nodes) - \text{CHOOSE } i \in \text{DOMAIN } m.backups : m.backups[i] =$$

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.

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

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* ≤ 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.

$$\begin{array}{l}
SendMasterArbitrationUpdate(n) \triangleq \\
\quad \land streams[n].state = Open \\
\quad \land \text{LET } m \triangleq mastership[n] \\
\qquad\quad s \triangleq streams[n] \\
\qquad \text{IN} \\
\qquad\quad \land m.term > 0 \\
\qquad\quad \land s.term < m.term \\
\qquad\quad \land \lor \land m.master = n
\end{array}$$

$$
\begin{array}{l}
\quad \land \mathit{SendRequest}(n, [\\
\qquad \mathit{type} \quad\quad \mapsto \mathit{MasterArbitrationUpdate}, \\
\qquad \mathit{election\_id} \mapsto \mathit{MasterElectionId}(m), \\
\qquad \mathit{term} \quad\quad \mapsto m.\mathit{term}]) \\
\lor \land m.\mathit{master} \neq n \\
\quad \land n \in \mathit{Range}(m.\mathit{backups}) \\
\quad \land \mathit{SendRequest}(n, [\\
\qquad \mathit{type} \quad\quad \mapsto \mathit{MasterArbitrationUpdate}, \\
\qquad \mathit{election\_id} \mapsto \mathit{BackupElectionId}(n,\ m), \\
\qquad \mathit{term} \quad\quad \mapsto m.\mathit{term}]) \\
\land \mathit{streams}' = [\mathit{streams} \text{ EXCEPT } ![n].\mathit{term} = m.\mathit{term}] \\
\land \text{UNCHANGED } \langle \mathit{mastershipVars},\ \mathit{events},\ \mathit{masterships},\ \mathit{isMaster},\ \mathit{streamChanges},\ \mathit{responses} \rangle
\end{array}
$$

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.

$$
\begin{array}{l}
\mathit{ReceiveMasterArbitrationUpdate}(n) \triangleq \\
\quad \land \mathit{streams}[n].\mathit{state} = \mathit{Open} \\
\quad \land \mathit{HasResponse}(n,\ \mathit{MasterArbitrationUpdate}) \\
\quad \land \text{LET } r \triangleq \mathit{NextResponse}(n) \\
\qquad\quad m \triangleq \mathit{masterships}[n] \\
\qquad\quad s \triangleq \mathit{streams}[n] \\
\qquad \text{IN} \\
\qquad\quad \lor \land r.\mathit{status} = \mathit{Ok} \\
\qquad\qquad \land m.\mathit{master} = n \\
\qquad\qquad \land m.\mathit{term} = \mathit{MasterTerm}(r) \\
\qquad\qquad \land s.\mathit{term} = m.\mathit{term} \\
\qquad\qquad \land \mathit{isMaster}' = [\mathit{isMaster} \text{ EXCEPT } ![n] = \text{TRUE}] \\
\qquad\quad \lor \land \lor r.\mathit{status} \neq \mathit{Ok} \\
\qquad\qquad\quad \lor m.\mathit{master} \neq n \\
\qquad\qquad\quad \lor s.\mathit{term} \neq m.\mathit{term} \\
\qquad\qquad\quad \lor m.\mathit{term} \neq \mathit{MasterTerm}(r) \\
\qquad\qquad \land \mathit{isMaster}' = [\mathit{isMaster} \text{ EXCEPT } ![n] = \text{FALSE}] \\
\quad \land \mathit{DiscardResponse}(n) \\
\quad \land \text{UNCHANGED } \langle \mathit{events},\ \mathit{masterships},\ \mathit{mastershipVars},\ \mathit{streamVars},\ \mathit{requests},\ \mathit{messageCount} \rangle
\end{array}
$$

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 'masterships') 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) ≜
    ∧ streams[n].state = Open
    ∧ LET m ≜ masterships[n]
      IN
          ∧ m.term > 0
          ∧ m.master = n
          ∧ isMaster[n]
          ∧ SendRequest(n, [
                type        ↦ WriteRequest,
                election_id ↦ MasterElectionId(m),
                term        ↦ m.term])
    ∧ UNCHANGED ⟨mastershipVars, nodeVars, streamVars, responses⟩

 Node 'n' receives a write response from the device
ReceiveWriteResponse(n) ≜
    ∧ streams[n].state = Open
    ∧ HasResponse(n, WriteResponse)
    ∧ LET m ≜ NextResponse(n)
      IN
          ∨ m.status = Ok
          ∨ m.status = PermissionDenied
    ∧ DiscardResponse(n)
    ∧ UNCHANGED ⟨mastershipVars, nodeVars, streamVars, requests, messageCount⟩
```

* Modification History
* Last modified *Wed Feb* 20 23:59:46 *PST* 2019 by *jordanhalterman*
* Created *Wed Feb* 20 23:49:08 *PST* 2019 by *jordanhalterman*