MODULE *P4RuntimeElection*

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

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

Stream states
CONSTANTS *Open*, *Closed*

Master arbitration message types
CONSTANTS *MasterArbitrationUpdate*

Write message types
CONSTANTS *WriteRequest*, *WriteResponse*

Response status constants
CONSTANTS *Ok*, *AlreadyExists*, *PermissionDenied*

Empty value
CONSTANT *Nil*

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*

The state of all streams and their requests and responses
VARIABLE *streams*, *requests*, *responses*

The current set of elections for the switch, the greatest of which is the current master
VARIABLE *elections*

Counting variables used to enforce state constraints
VARIABLES *mastershipChanges*, *streamChanges*, *messageCount*

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

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

Stream related variables
$streamVars \triangleq \langle streams, streamChanges \rangle$

Message related variables

$messageVars \triangleq \langle requests,\ responses,\ messageCount \rangle$

Device related variables

$deviceVars \triangleq \langle elections \rangle$

A sequence of all variables

$vars \triangleq \langle mastershipVars,\ nodeVars,\ streamVars,\ messageVars,\ deviceVars \rangle$

Helpers

Returns a sequence with the head removed

$Pop(q) \triangleq SubSeq(q,\ 2,\ Len(q))$

Returns a sequences with the element at the given index removed

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

Returns the set of values in f

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

Returns the maximum value from a set or undefined if the set is empty

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

Messaging between the *Nodes* and the device are modelled on *TCP*. For each node, a request and response sequence provides ordered messaging between the two points. Requests and responses are always received from the head of the queue and are never duplicated or reordered, and request and response queues only last the lifetime of the stream. When a stream is closed, all that stream's requests and responses are lost.

Sends request 'm' on the stream for node 'n'

$SendRequest(n,\ m) \triangleq$

$\quad \land\ requests' = [requests \text{ EXCEPT } ![n] = Append(requests[n],\ m)]$

$\quad \land\ messageCount' = messageCount + 1$

Indicates whether any requests are in the queue for node 'n'

$HasRequest(n,\ t) \triangleq Len(requests[n]) > 0 \land requests[n][1].type = t$

Returns the next request in the queue for node 'n'

$NextRequest(n) \triangleq requests[n][1]$

Discards the request at the head of the queue for node 'n'

$DiscardRequest(n) \triangleq requests' = [requests \text{ EXCEPT } ![n] = Pop(requests[n])]$

Sends response 'm' on the stream for node 'n'

$SendResponse(n,\ m) \triangleq$

$\quad \land\ responses' = [responses \text{ EXCEPT } ![n] = Append(responses[n],\ m)]$

$\quad \land\ messageCount' = messageCount + 1$

Indicates whether any responses are in the queue for node 'n'

$$HasResponse(n,\ t) \triangleq Len(responses[n]) > 0 \land responses[n][1].type = t$$

Returns the next response in the queue for node 'n'

$$NextResponse(n) \triangleq responses[n][1]$$

Discards the response at the head of the queue for node 'n'

$$DiscardResponse(n) \triangleq responses' = [responses \text{ EXCEPT } ![n] = Pop(responses[n])]$$

This section models mastership arbitration on the controller side. Mastership election occurs in two disctinct types of state changes. One state change occurs to change the mastership in the consensus layer, and the other occurs when a node actually learns of the mastership change. Nodes will always learn of mastership changes in the order in which they occur, and nodes will always learn of a mastership change. This, of course, is not representative of practice but is sufficient for modelling the mastership election algorithm.

Adds a node to the mastership election

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

Removes a node from the mastership election

$$\begin{array}{l}
LeaveMastershipElection(n) \triangleq \\
\quad \land \lor \land master = n \\
\qquad \land \lor \land Len(backups) > 0 \\
\qquad\quad \land term' = term + 1 \\
\qquad\quad \land master' = backups[1] \\
\qquad\quad \land backups' = Pop(backups) \\
\qquad\quad \land events' = [i \quad \in Nodes \mapsto Append(events[i], [
\end{array}$$

```
                                                    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 ⟨masterships, streamVars, messageVars, deviceVars⟩

Sets the current master to node 'n' if it's not already set
SetMastership(n) ≜
    ∨ ∧ master = n
      ∧ UNCHANGED ⟨mastershipVars⟩
    ∨ ∧ master ≠ n
      ∧ term′ = term + 1
      ∧ master′ = n
      ∧ ∨ ∧ n ∈ Range(backups)
          ∧ backups′ = Drop(backups, CHOOSE j ∈ DOMAIN backups : backups[j] = n)
        ∨ ∧ n ∉ Range(backups)
          ∧ UNCHANGED ⟨backups⟩
      ∧ mastershipChanges′ = mastershipChanges + 1

Receives a mastership change event from the consensus layer on node 'n'
LearnMastership(n) ≜
    ∧ Len(events[n]) > 0
    ∧ LET e ≜ events[n][1]
          m ≜ masterships[n]
      IN
          ∨ ∧ e.term > m.term
            ∧ masterships′ = [masterships EXCEPT ![n] = [
                                 term   ↦ e.term,
                                 master ↦ e.master,
                                 backups ↦ e.backups,
                                 sent   ↦ FALSE]]
          ∨ ∧ e.term = m.term
            ∧ masterships′ = [masterships EXCEPT ![n] = [
                                 term   ↦ e.term,
                                 master ↦ e.master,
                                 backups ↦ e.backups,
                                 sent   ↦ m.sent]]
    ∧ events′ = [events EXCEPT ![n] = Pop(events[n])]
    ∧ UNCHANGED ⟨mastershipVars, streamVars, messageVars, deviceVars⟩
```

```
 Notifies the device of node 'n' mastership info if it hasn't already been sent
SendMasterArbitrationUpdateRequest(n) ≜
    ∧ streams[n] = Open
    ∧ LET m ≜ masterships[n]
      IN
          ∧ m.term > 0
          ∧ ¬m.sent
          ∧ ∨ ∧ m.master = n
              ∧ SendRequest(n, [
                    type         ↦ MasterArbitrationUpdate,
                    election_id ↦ m.term + Cardinality(Nodes)])
            ∨ ∧ m.master ≠ n
              ∧ n ∈ Range(m.backups)
              ∧ SendRequest(n, [
                    type         ↦ MasterArbitrationUpdate,
                    election_id ↦ m.term + Cardinality(Nodes) − CHOOSE i ∈ DOMAIN m.backups : m.backu
    ∧ masterships' = [masterships EXCEPT ![n].sent = TRUE]
    ∧ UNCHANGED ⟨mastershipVars, events, deviceVars, streamVars, responses⟩

 Receives a master arbitration update response on node 'n'
ReceiveMasterArbitrationUpdateResponse(n) ≜
    ∧ streams[n] = Open
    ∧ HasResponse(n, MasterArbitrationUpdate)
    ∧ LET m ≜ NextResponse(n)
      IN
          ∨ ∧ m.status = Ok
            ∧ SetMastership(n)
          ∨ ∧ m.status = AlreadyExists
            ∧ UNCHANGED ⟨mastershipVars⟩
    ∧ DiscardResponse(n)
    ∧ UNCHANGED ⟨nodeVars, deviceVars, streamVars, requests, messageCount⟩

 Sends a write request to the device from node 'n'
SendWriteRequest(n) ≜
    ∧ streams[n] = Open
    ∧ LET m ≜ masterships[n]
      IN
          ∧ m.term > 0
          ∧ m.master = n
          ∧ SendRequest(n, [
                type         ↦ WriteRequest,
                election_id ↦ m.term + Cardinality(Nodes)])
    ∧ UNCHANGED ⟨mastershipVars, nodeVars, deviceVars, streamVars, responses⟩

 Receives a write response on node 'n'
ReceiveWriteResponse(n) ≜
```

$\land$ $streams[n] = Open$
$\land$ $HasResponse(n,\ WriteResponse)$
$\land$ LET $m \triangleq NextResponse(n)$
IN
TODO: This should be used to determine whether writes from old masters are allowed
$\lor$ $m.status = Ok$
$\lor$ $m.status = PermissionDenied$
$\land$ $DiscardResponse(n)$
$\land$ UNCHANGED $\langle mastershipVars,\ nodeVars,\ deviceVars,\ streamVars,\ requests,\ messageCount\rangle$

This section models the $P4$ switch. The switch side manages stream states between the device and the controller. Streams are opened and closed in a single state transition for the purposes of this model. Switches can handle two types of messages from the controller nodes: *MasterArbitrationUpdate* and Write.

Returns the highest election *ID* for the given elections
$ElectionId(e) \triangleq Max(Range(e))$

Returns the master for the given elections
$Master(e) \triangleq$
IF $Cardinality(\{i \in Range(e) : i > 0\}) > 0$ THEN
CHOOSE $n \in$ DOMAIN $e : e[n] = ElectionId(e)$
ELSE
Nil

Opens a new stream between node 'n' and the device
When a new stream is opened, the 'requests' and 'responses' queues for the node are cleared and the 'streams' state is set to 'Open'.
$ConnectStream(n) \triangleq$
$\land$ $streams[n] = Closed$
$\land$ $streams' = [streams$ EXCEPT $![n] = Open]$
$\land$ $requests' = [requests$ EXCEPT $![n] = \langle\rangle]$
$\land$ $responses' = [responses$ EXCEPT $![n] = \langle\rangle]$
$\land$ $streamChanges' = streamChanges + 1$
$\land$ UNCHANGED $\langle mastershipVars,\ nodeVars,\ deviceVars,\ messageCount\rangle$

Closes the open stream between node 'n' and the device
When the stream is closed, the 'requests' and 'responses' queues for the node are cleared and a 'MasterArbitrationUpdate' is sent to all remaining connected nodes to notify them of a mastership change if necessary.
$CloseStream(n) \triangleq$
$\land$ $streams[n] = Open$
$\land$ $elections' = [elections$ EXCEPT $![n] = 0]$
$\land$ $streams' = [streams$ EXCEPT $![n] = Closed]$
$\land$ $requests' = [requests$ EXCEPT $![n] = \langle\rangle]$
$\land$ LET $oldMaster \triangleq Master(elections)$

```
        newMaster ≜ Master(elections')
    IN
        ∨ ∧ oldMaster ≠ newMaster
          ∧ responses' = [i ∈ DOMAIN streams' ↦
                            IF i = newMaster THEN
                                Append(responses[i], [
                                    type        ↦ MasterArbitrationUpdate,
                                    status      ↦ Ok,
                                    election_id ↦ ElectionId(elections')])
                             ELSE
                                Append(responses[i], [
                                    type        ↦ MasterArbitrationUpdate,
                                    status      ↦ AlreadyExists,
                                    election_id ↦ ElectionId(elections')])]
          ∧ messageCount' = messageCount + 1
        ∨ ∧ oldMaster = newMaster
          ∧ responses' = [responses EXCEPT ![n] = ⟨⟩]
          ∧ UNCHANGED ⟨messageCount⟩
    ∧ streamChanges' = streamChanges + 1
    ∧ UNCHANGED ⟨mastershipVars, nodeVars⟩

Handles a master arbitration update on the device
If the election_id is already present in the 'elections', send an 'AlreadyExists'
response to the node. Otherwise,
HandleMasterArbitrationUpdate(n) ≜
    ∧ streams[n] = Open
    ∧ HasRequest(n, MasterArbitrationUpdate)
    ∧ LET m ≜ NextRequest(n)
      IN
        ∨ ∧ m.election_id ∈ Range(elections)
          ∧ SendResponse(n, [
                type        ↦ MasterArbitrationUpdate,
                election_id ↦ m.election_id,
                status      ↦ AlreadyExists])
          ∧ UNCHANGED ⟨deviceVars⟩
        ∨ ∧ m.election_id ∉ Range(elections)
          ∧ elections' = [elections EXCEPT ![n] = m.election_id]
          ∧ LET oldMaster ≜ Master(elections)
                newMaster ≜ Master(elections')
            IN
              ∨ ∧ oldMaster ≠ newMaster
                ∧ responses' = [i ∈ DOMAIN streams ↦
                                  IF i = newMaster THEN
                                      Append(responses[i], [
                                          type        ↦ MasterArbitrationUpdate,
```

```
                                        status       ↦ Ok,
                                        election_id ↦ ElectionId(elections')])
                                ELSE
                                    Append(responses[i], [
                                        type         ↦ MasterArbitrationUpdate,
                                        status       ↦ AlreadyExists,
                                        election_id ↦ ElectionId(elections')])]
                        ∧ messageCount' = messageCount + 1
                    ∨ ∧ oldMaster = newMaster
                      ∧ SendResponse(n, [
                            type         ↦ MasterArbitrationUpdate,
                            status       ↦ Ok,
                            election_id ↦ ElectionId(elections')])
    ∧ DiscardRequest(n)
    ∧ UNCHANGED ⟨mastershipVars, nodeVars, streamVars⟩

Handles a write request on the device
HandleWrite(n) ≜
    ∧ streams[n] = Open
    ∧ HasRequest(n, WriteRequest)
    ∧ LET m ≜ NextRequest(n)
      IN
          ∨ ∧ Cardinality(DOMAIN elections) = 0
            ∧ SendResponse(n, [
                  type   ↦ WriteResponse,
                  status ↦ PermissionDenied])
          ∨ ∧ ElectionId(elections) ≠ m.election_id
            ∧ SendResponse(n, [
                  type   ↦ WriteResponse,
                  status ↦ PermissionDenied])
          ∨ ∧ m.election_id ∉ Range(elections)
            ∧ elections[n] = m.election_id
            ∧ SendResponse(n, [
                  type   ↦ WriteResponse,
                  status ↦ Ok])
    ∧ DiscardRequest(n)
    ∧ UNCHANGED ⟨mastershipVars, nodeVars, deviceVars, streamVars⟩

─────────────────────────────────────────────────────────────────────

Init ≜
    ∧ term = 0
    ∧ master = Nil
    ∧ backups = ⟨⟩
    ∧ events = [n ∈ Nodes ↦ ⟨⟩]
    ∧ masterships = [n ∈ Nodes ↦ [term ↦ 0, master ↦ 0, backups ↦ ⟨⟩, sent ↦ FALSE]]
```

$\land\ streams = [n \in Nodes \mapsto Closed]$
$\land\ requests = [n \in Nodes \mapsto \langle\rangle]$
$\land\ responses = [n \in Nodes \mapsto \langle\rangle]$
$\land\ elections\ = [n \in Nodes \mapsto 0]$
$\land\ mastershipChanges = 0$
$\land\ streamChanges = 0$
$\land\ messageCount = 0$

$Next\ \triangleq$
$\lor \exists\, n \in Nodes : ConnectStream(n)$
$\lor \exists\, n \in Nodes : CloseStream(n)$
$\lor \exists\, n \in Nodes : JoinMastershipElection(n)$
$\lor \exists\, n \in Nodes : LeaveMastershipElection(n)$
$\lor \exists\, n \in Nodes : LearnMastership(n)$
$\lor \exists\, n \in Nodes : SendMasterArbitrationUpdateRequest(n)$
$\lor \exists\, n \in Nodes : HandleMasterArbitrationUpdate(n)$
$\lor \exists\, n \in Nodes : ReceiveMasterArbitrationUpdateResponse(n)$
$\lor \exists\, n \in Nodes : SendWriteRequest(n)$
$\lor \exists\, n \in Nodes : HandleWrite(n)$
$\lor \exists\, n \in Nodes : ReceiveWriteResponse(n)$

$Spec\ \triangleq\ Init \land \Box[Next]_{vars}$

* Modification History
* Last modified Sun *Feb* 17 00:35:14 *PST* 2019 by *jordanhalterman*
* Created *Thu Feb* 14 11:33:03 *PST* 2019 by *jordanhalterman*