

---

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*

A sequence of successful writes to the switch used for model checking

VARIABLE *writes*

---

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  
 $deviceVars \triangleq \langle elections, writes \rangle$

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'  
 $SendRequest(n, m) \triangleq$   
 $\wedge requests' = [requests \text{ EXCEPT } ![n] = Append(requests[n], m)]$   
 $\wedge messageCount' = messageCount + 1$

Indicates whether any requests are in the queue for node 'n'  
 $HasRequest(n, t) \triangleq Len(requests[n]) > 0 \wedge requests[n][1].type = t$

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'  
 $SendResponse(n, m) \triangleq$

$$\begin{aligned} &\wedge responses' = [responses \text{ EXCEPT } ![n] = Append(responses[n], m)] \\ &\wedge messageCount' = messageCount + 1 \end{aligned}$$

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

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

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 distinct 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

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

Removes a node from the mastership election

$$\begin{aligned} LeaveMastershipElection(n) \triangleq & \\ &\wedge \vee \wedge master = n \\ &\quad \wedge \vee \wedge Len(backups) > 0 \\ &\quad \quad \wedge term' = term + 1 \\ &\quad \quad \wedge master' = backups[1] \end{aligned}$$

$$\begin{aligned}
& \wedge \text{backups}' = \text{Pop}(\text{backups}) \\
& \wedge \text{events}' = [i \in \text{Nodes} \mapsto \text{Append}(\text{events}[i], [ \\
& \qquad \qquad \qquad \text{term} \mapsto \text{term}', \\
& \qquad \qquad \qquad \text{master} \mapsto \text{master}', \\
& \qquad \qquad \qquad \text{backups} \mapsto \text{backups'}])] \\
& \vee \wedge \text{Len}(\text{backups}) = 0 \\
& \quad \wedge \text{master}' = \text{Nil} \\
& \quad \wedge \text{UNCHANGED } \langle \text{term}, \text{backups}, \text{events} \rangle \\
& \vee \wedge n \in \text{Range}(\text{backups}) \\
& \quad \wedge \text{backups}' = \text{Drop}(\text{backups}, \text{CHOOSE } j \in \text{DOMAIN } \text{backups} : \text{backups}[j] = n) \\
& \quad \wedge \text{UNCHANGED } \langle \text{term}, \text{master}, \text{events} \rangle \\
& \wedge \text{mastershipChanges}' = \text{mastershipChanges} + 1 \\
& \wedge \text{UNCHANGED } \langle \text{masterships}, \text{streamVars}, \text{messageVars}, \text{deviceVars} \rangle
\end{aligned}$$

Sets the current master to node 'n' if it's not already set

$$\begin{aligned}
\text{SetMastership}(n) & \triangleq \\
& \vee \wedge \text{master} = n \\
& \quad \wedge \text{UNCHANGED } \langle \text{mastershipVars} \rangle \\
& \vee \wedge \text{master} \neq n \\
& \quad \wedge \text{term}' = \text{term} + 1 \\
& \quad \wedge \text{master}' = n \\
& \quad \wedge \vee \wedge n \in \text{Range}(\text{backups}) \\
& \quad \quad \wedge \text{backups}' = \text{Drop}(\text{backups}, \text{CHOOSE } j \in \text{DOMAIN } \text{backups} : \text{backups}[j] = n) \\
& \quad \vee \wedge n \notin \text{Range}(\text{backups}) \\
& \quad \quad \wedge \text{UNCHANGED } \langle \text{backups} \rangle \\
& \quad \wedge \text{mastershipChanges}' = \text{mastershipChanges} + 1
\end{aligned}$$

Receives a mastership change event from the consensus layer on node 'n'

$$\begin{aligned}
\text{LearnMastership}(n) & \triangleq \\
& \wedge \text{Len}(\text{events}[n]) > 0 \\
& \wedge \text{LET } e \triangleq \text{events}[n][1] \\
& \quad m \triangleq \text{masterships}[n] \\
& \text{IN} \\
& \quad \vee \wedge e.\text{term} > m.\text{term} \\
& \quad \quad \wedge \text{masterships}' = [\text{masterships} \text{ EXCEPT } ![n] = [ \\
& \qquad \qquad \qquad \text{term} \mapsto e.\text{term}, \\
& \qquad \qquad \qquad \text{master} \mapsto e.\text{master}, \\
& \qquad \qquad \qquad \text{backups} \mapsto e.\text{backups}, \\
& \qquad \qquad \qquad \text{sent} \mapsto \text{FALSE}]] \\
& \quad \vee \wedge e.\text{term} = m.\text{term} \\
& \quad \quad \wedge \text{masterships}' = [\text{masterships} \text{ EXCEPT } ![n] = [ \\
& \qquad \qquad \qquad \text{term} \mapsto e.\text{term}, \\
& \qquad \qquad \qquad \text{master} \mapsto e.\text{master}, \\
& \qquad \qquad \qquad \text{backups} \mapsto e.\text{backups}, \\
& \qquad \qquad \qquad \text{sent} \mapsto m.\text{sent}]
\end{aligned}$$

$\wedge events' = [events \text{ EXCEPT } ![n] = Pop(events[n])]$   
 $\wedge \text{UNCHANGED } \langle mastershipVars, streamVars, messageVars, deviceVars \rangle$

Notifies the device of node 'n' mastership info if it hasn't already been sent

$SendMasterArbitrationUpdateRequest(n) \triangleq$

$\wedge streams[n] = Open$

$\wedge \text{LET } m \triangleq masterships[n]$

IN

$\wedge m.term > 0$

$\wedge \neg m.sent$

$\wedge \vee \wedge m.master = n$

$\wedge SendRequest(n, [$   
 $\quad type \quad \mapsto MasterArbitrationUpdate,$   
 $\quad election\_id \mapsto m.term + Cardinality(Nodes),$   
 $\quad term \quad \mapsto m.term])$

$\vee \wedge m.master \neq n$

$\wedge n \in Range(m.backups)$

$\wedge SendRequest(n, [$   
 $\quad type \quad \mapsto MasterArbitrationUpdate,$   
 $\quad election\_id \mapsto m.term + Cardinality(Nodes) - \text{CHOOSE } i \in \text{DOMAIN } m.backups : m.backups[i],$   
 $\quad term \quad \mapsto m.term])$

$\wedge masterships' = [masterships \text{ EXCEPT } ![n].sent = \text{TRUE}]$

$\wedge \text{UNCHANGED } \langle mastershipVars, events, deviceVars, streamVars, responses \rangle$

Receives a master arbitration update response on node 'n'

$ReceiveMasterArbitrationUpdateResponse(n) \triangleq$

$\wedge streams[n] = Open$

$\wedge HasResponse(n, MasterArbitrationUpdate)$

$\wedge \text{LET } m \triangleq NextResponse(n)$

IN

$\vee \wedge m.status = Ok$

$\wedge SetMastership(n)$

$\vee \wedge m.status = AlreadyExists$

$\wedge \text{UNCHANGED } \langle mastershipVars \rangle$

$\wedge DiscardResponse(n)$

$\wedge \text{UNCHANGED } \langle nodeVars, deviceVars, streamVars, requests, messageCount \rangle$

Sends a write request to the device from node 'n'

$SendWriteRequest(n) \triangleq$

$\wedge streams[n] = Open$

$\wedge \text{LET } m \triangleq masterships[n]$

IN

$\wedge m.term > 0$

$\wedge m.master = n$

$\wedge SendRequest(n, [$   
 $\quad type \quad \mapsto WriteRequest,$

$$\begin{aligned}
& \text{election\_id} \mapsto m.\text{term} + \text{Cardinality}(\text{Nodes}), \\
& \text{term} \mapsto m.\text{term}] \\
& \wedge \text{UNCHANGED } \langle \text{mastershipVars}, \text{nodeVars}, \text{deviceVars}, \text{streamVars}, \text{responses} \rangle
\end{aligned}$$

Receives a write response on node 'n'

$$\begin{aligned}
\text{ReceiveWriteResponse}(n) & \triangleq \\
& \wedge \text{streams}[n] = \text{Open} \\
& \wedge \text{HasResponse}(n, \text{WriteResponse}) \\
& \wedge \text{LET } m \triangleq \text{NextResponse}(n) \\
& \text{IN} \\
& \quad \text{TODO: This should be used to determine whether writes from old masters are allowed} \\
& \quad \vee m.\text{status} = \text{Ok} \\
& \quad \vee m.\text{status} = \text{PermissionDenied} \\
& \wedge \text{DiscardResponse}(n) \\
& \wedge \text{UNCHANGED } \langle \text{mastershipVars}, \text{nodeVars}, \text{deviceVars}, \text{streamVars}, \text{requests}, \text{messageCount} \rangle
\end{aligned}$$


---

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

$$\text{ElectionId}(e) \triangleq \text{Max}(\text{Range}(e))$$

Returns the master for the given elections

$$\begin{aligned}
\text{Master}(e) & \triangleq \\
& \text{IF } \text{Cardinality}(\{i \in \text{Range}(e) : i > 0\}) > 0 \text{ THEN} \\
& \quad \text{CHOOSE } n \in \text{DOMAIN } e : e[n] = \text{ElectionId}(e) \\
& \text{ELSE} \\
& \quad \text{Nil}
\end{aligned}$$

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'.

$$\begin{aligned}
\text{ConnectStream}(n) & \triangleq \\
& \wedge \text{streams}[n] = \text{Closed} \\
& \wedge \text{streams}' = [\text{streams} \text{ EXCEPT } ![n] = \text{Open}] \\
& \wedge \text{requests}' = [\text{requests} \text{ EXCEPT } ![n] = \langle \rangle] \\
& \wedge \text{responses}' = [\text{responses} \text{ EXCEPT } ![n] = \langle \rangle] \\
& \wedge \text{streamChanges}' = \text{streamChanges} + 1 \\
& \wedge \text{UNCHANGED } \langle \text{mastershipVars}, \text{nodeVars}, \text{deviceVars}, \text{messageCount} \rangle
\end{aligned}$$

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.

$$\begin{aligned}
& \text{CloseStream}(n) \triangleq \\
& \quad \wedge \text{streams}[n] = \text{Open} \\
& \quad \wedge \text{elections}' = [\text{elections} \text{ EXCEPT } ![n] = 0] \\
& \quad \wedge \text{streams}' = [\text{streams} \text{ EXCEPT } ![n] = \text{Closed}] \\
& \quad \wedge \text{requests}' = [\text{requests} \text{ EXCEPT } ![n] = \langle \rangle] \\
& \quad \wedge \text{LET } \text{oldMaster} \triangleq \text{Master}(\text{elections}) \\
& \quad \quad \text{newMaster} \triangleq \text{Master}(\text{elections}') \\
& \quad \text{IN} \\
& \quad \quad \vee \wedge \text{oldMaster} \neq \text{newMaster} \\
& \quad \quad \quad \wedge \text{responses}' = [i \in \text{DOMAIN } \text{streams}' \mapsto \\
& \quad \quad \quad \quad \text{IF } i = \text{newMaster} \text{ THEN} \\
& \quad \quad \quad \quad \quad \text{Append}(\text{responses}[i], [ \\
& \quad \quad \quad \quad \quad \quad \text{type} \quad \mapsto \text{MasterArbitrationUpdate}, \\
& \quad \quad \quad \quad \quad \quad \text{status} \quad \mapsto \text{Ok}, \\
& \quad \quad \quad \quad \quad \quad \text{election\_id} \mapsto \text{ElectionId}(\text{elections}')] \\
& \quad \quad \quad \quad \text{ELSE} \\
& \quad \quad \quad \quad \quad \text{Append}(\text{responses}[i], [ \\
& \quad \quad \quad \quad \quad \quad \text{type} \quad \mapsto \text{MasterArbitrationUpdate}, \\
& \quad \quad \quad \quad \quad \quad \text{status} \quad \mapsto \text{AlreadyExists}, \\
& \quad \quad \quad \quad \quad \quad \text{election\_id} \mapsto \text{ElectionId}(\text{elections}')] \\
& \quad \quad \quad \quad \wedge \text{messageCount}' = \text{messageCount} + 1 \\
& \quad \quad \quad \vee \wedge \text{oldMaster} = \text{newMaster} \\
& \quad \quad \quad \quad \wedge \text{responses}' = [\text{responses} \text{ EXCEPT } ![n] = \langle \rangle] \\
& \quad \quad \quad \quad \wedge \text{UNCHANGED } \langle \text{messageCount} \rangle \\
& \quad \quad \wedge \text{streamChanges}' = \text{streamChanges} + 1 \\
& \quad \quad \wedge \text{UNCHANGED } \langle \text{mastershipVars}, \text{nodeVars}, \text{writes} \rangle
\end{aligned}$$

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,

$$\begin{aligned}
& \text{HandleMasterArbitrationUpdate}(n) \triangleq \\
& \quad \wedge \text{streams}[n] = \text{Open} \\
& \quad \wedge \text{HasRequest}(n, \text{MasterArbitrationUpdate}) \\
& \quad \wedge \text{LET } m \triangleq \text{NextRequest}(n) \\
& \quad \text{IN} \\
& \quad \quad \vee \wedge m.\text{election\_id} \in \text{Range}(\text{elections}) \\
& \quad \quad \quad \wedge \text{SendResponse}(n, [ \\
& \quad \quad \quad \quad \text{type} \quad \mapsto \text{MasterArbitrationUpdate}, \\
& \quad \quad \quad \quad \text{election\_id} \mapsto m.\text{election\_id}, \\
& \quad \quad \quad \quad \text{status} \quad \mapsto \text{AlreadyExists}] \\
& \quad \quad \quad \wedge \text{UNCHANGED } \langle \text{deviceVars} \rangle \\
& \quad \quad \vee \wedge m.\text{election\_id} \notin \text{Range}(\text{elections}) \\
& \quad \quad \quad \wedge \text{elections}' = [\text{elections} \text{ EXCEPT } ![n] = m.\text{election\_id}] \\
& \quad \quad \quad \wedge \text{LET } \text{oldMaster} \triangleq \text{Master}(\text{elections}) \\
& \quad \quad \quad \quad \text{newMaster} \triangleq \text{Master}(\text{elections}')
\end{aligned}$$

```

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, writes⟩

```

Handles a write request on the device

```

HandleWrite(n) ≜
  ∧ streams[n] = Open
  ∧ HasRequest(n, WriteRequest)
  ∧ LET m ≜ NextRequest(n)
  IN
    ∨ ∧ Master(elections) ≠ n
      ∧ SendResponse(n, [
        type      ↦ WriteResponse,
        status    ↦ PermissionDenied])
      ∧ UNCHANGED ⟨writes⟩
    ∨ ∧ Master(elections) = n
      ∧ writes' = Append(writes, [node ↦ n, term ↦ m.term])
      ∧ SendResponse(n, [
        type      ↦ WriteResponse,
        status    ↦ Ok])
    ∧ DiscardRequest(n)
  ∧ UNCHANGED ⟨mastershipVars, nodeVars, elections, streamVars⟩

```

The invariant asserts that no master can write to the switch after the switch has been notified of a newer master

*TypeInvariant* ≜ ∀ *i* ∈ DOMAIN *writes* : *i* = 1 ∨ *writes*[*i* - 1].*term* ≤ *writes*[*i*].*term*



---

$Init \triangleq$   
 $\wedge term = 0$   
 $\wedge master = Nil$   
 $\wedge backups = \langle \rangle$   
 $\wedge events = [n \in Nodes \mapsto \langle \rangle]$   
 $\wedge masterships = [n \in Nodes \mapsto [term \mapsto 0, master \mapsto 0, backups \mapsto \langle \rangle, sent \mapsto FALSE]]$   
 $\wedge streams = [n \in Nodes \mapsto Closed]$   
 $\wedge requests = [n \in Nodes \mapsto \langle \rangle]$   
 $\wedge responses = [n \in Nodes \mapsto \langle \rangle]$   
 $\wedge elections = [n \in Nodes \mapsto 0]$   
 $\wedge mastershipChanges = 0$   
 $\wedge streamChanges = 0$   
 $\wedge messageCount = 0$   
 $\wedge writes = \langle \rangle$

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

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

---

$\backslash$  \* Modification History  
 $\backslash$  \* Last modified Sun Feb 17 01:26:56 PST 2019 by *jordanhalterman*  
 $\backslash$  \* Created Thu Feb 14 11:33:03 PST 2019 by *jordanhalterman*