MODULE *P4RuntimeRevision*

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

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

Stream states
CONSTANTS *Open*, *Closed*

Write message types
CONSTANTS *WriteRequest*, *WriteResponse*

Response status constants
CONSTANTS *Ok*, *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 term of the last successful write to the device
VARIABLE *lastTerm*

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

A history of successful writes to the switch used for model checking
VARIABLE *history*

---

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

$$deviceVars \triangleq \langle lastTerm,\ history \rangle$$

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

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

---

This section models the messaging between controller nodes and the device. Messaging is modelled on *TCP*, providing strict ordering between controller and device via sequences. The 'requests' sequence represents the messages from controller to device for each node, and the 'responses' sequence represents the messages from device to each node. Requests and responses are always received from the head of the queue and are never duplicated or reordered.

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 a request of type 't' is at the head of 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 a response of type 't' is at the head of 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'

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

Indicates whether the stream for node 'n' is *Open*

$$IsStreamOpen(n) \triangleq streams[n].state = Open$$

Indicates whether the stream for node 'n' is *Closed*

$$IsStreamClosed(n) \triangleq streams[n].state = Closed$$

---

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.

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'.

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

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.

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

---

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.

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.

$$
\begin{array}{l}
LearnMastership(n) \triangleq \\
\quad \land Len(events[n]) > 0 \\
\quad \land \text{LET } e \triangleq events[n][1] \\
\qquad\quad\ \ m \triangleq masterships[n] \\
\quad\ \ \text{IN} \\
\qquad \lor \land e.term > m.term \\
\qquad\quad \land masterships' = [masterships \text{ EXCEPT } ![n] = [ \\
\qquad\qquad\qquad\qquad\qquad\qquad term \mapsto e.term, \\
\qquad\qquad\qquad\qquad\qquad\qquad master \mapsto e.master, \\
\qquad\qquad\qquad\qquad\qquad\qquad backups \mapsto e.backups]] \\
\qquad \lor \land e.term = m.term \\
\qquad\quad \land masterships' = [masterships \text{ EXCEPT } ![n] = [ \\
\qquad\qquad\qquad\qquad\qquad\qquad term \mapsto e.term, \\
\qquad\qquad\qquad\qquad\qquad\qquad master \mapsto e.master, \\
\qquad\qquad\qquad\qquad\qquad\qquad backups \mapsto e.backups]]
\end{array}
$$

$\land\ events' = [events \text{ EXCEPT } ![n] = Pop(events[n])]$
$\land$ UNCHANGED $\langle masterships Vars,\ stream Vars,\ message Vars,\ device Vars\rangle$

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.

$SendWriteRequest(n) \triangleq$
$\land\ IsStreamOpen(n)$
$\land$ LET $m \triangleq masterships[n]$
IN
$\land\ m.term > 0$
$\land\ m.master = n$
$\land\ SendRequest(n, [$
$type \mapsto WriteRequest,$
$term \mapsto m.term])$
$\land$ UNCHANGED $\langle masterships Vars,\ node Vars,\ device Vars,\ stream Vars,\ responses\rangle$

Node 'n' receives a write response from the device

$ReceiveWriteResponse(n) \triangleq$
$\land\ IsStreamOpen(n)$
$\land\ HasResponse(n,\ WriteResponse)$
$\land$ LET $m \triangleq NextResponse(n)$
IN
$\lor\ m.status = Ok$
$\lor\ m.status = PermissionDenied$
$\land\ DiscardResponse(n)$
$\land$ UNCHANGED $\langle masterships Vars,\ node Vars,\ device Vars,\ stream Vars,\ requests,\ messageCount\rangle$

---

This section models a *P*4 Runtime device. In this spec, the device's only role is to accept writes from controller nodes. Mastership order is maintained using a simple fencing token stored in 'lastTerm'. The device ensures only new masters can write to it by rejecting writes from older masters according to their provided term.

Opens a new stream between node 'n' and the device

When a stream is opened, the 'streams' state for node 'n' is set to *Open*. Stream creation is modelled as a single step to reduce the state space.

$ConnectStream(n) \triangleq$
$\land\ IsStreamClosed(n)$
$\land\ streams' = [streams \text{ EXCEPT } ![n].state = Open]$
$\land\ streamChanges' = streamChanges + 1$
$\land$ UNCHANGED $\langle masterships Vars,\ node Vars,\ device Vars,\ message Vars\rangle$

Closes an open stream between node 'n' and the device

When a stream is closed, the 'streams' state for node 'n' is set to *Closed*, and the 'requests' and 'responses' queues for the node are cleared.

$$
\begin{array}{l}
CloseStream(n) \triangleq \\
\quad \land\ IsStreamOpen(n) \\
\quad \land\ streams' = [streams \text{ EXCEPT } ![n] = [state \mapsto Closed,\ term \mapsto 0]] \\
\quad \land\ requests' = [requests \text{ EXCEPT } ![n] = \langle\rangle] \\
\quad \land\ responses' = [responses \text{ EXCEPT } ![n] = \langle\rangle] \\
\quad \land\ streamChanges' = streamChanges + 1 \\
\quad \land \text{UNCHANGED } \langle mastershipVars,\ nodeVars,\ deviceVars,\ messageCount\rangle
\end{array}
$$

The device receives a *WriteRequest* from node 'n'

If the *WriteRequest* 'term' is greater than or equal to the highest term received by the device, the write is accepted, the highest term is updated, and the write is recorded in history for model checking. Otherwise, the write was sent by an old master and is rejected with a *PermissionDenied* error.

$$
\begin{array}{l}
HandleWrite(n) \triangleq \\
\quad \land\ IsStreamOpen(n) \\
\quad \land\ HasRequest(n,\ WriteRequest) \\
\quad \land \text{LET } m \triangleq NextRequest(n) \\
\quad\ \text{IN} \\
\qquad \lor\ \land\ lastTerm \leq m.term \\
\qquad\quad \land\ lastTerm' = m.term \\
\qquad\quad \land\ history' = Append(history,\ [node \mapsto n,\ term \mapsto m.term]) \\
\qquad\quad \land\ SendResponse(n,\ [ \\
\qquad\qquad\quad type \mapsto WriteResponse, \\
\qquad\qquad\quad status \mapsto Ok]) \\
\qquad \lor\ \land\ lastTerm > m.term \\
\qquad\quad \land\ SendResponse(n,\ [ \\
\qquad\qquad\quad type \mapsto WriteResponse, \\
\qquad\qquad\quad status \mapsto PermissionDenied]) \\
\qquad\quad \land \text{UNCHANGED } \langle lastTerm,\ history\rangle \\
\quad \land\ DiscardRequest(n) \\
\quad \land \text{UNCHANGED } \langle mastershipVars,\ nodeVars,\ streamVars\rangle
\end{array}
$$

---

The invariant asserts that the device will not allow a write from an older master if it has already accepted a write from a newer master. This is determined by comparing the mastership terms of accepted writes. For this invariant to hold, terms may only increase in the history of writes.

$$
\begin{array}{l}
TypeInvariant \triangleq \\
\quad \land\ \forall\, x \in 1\,..\,Len(history) : \\
\qquad \forall\, y \in x\,..\,Len(history) : \\
\qquad\quad history[x].term \leq history[y].term \\
\quad \land\ \forall\, x \in 1\,..\,Len(history) : \\
\qquad \forall\, y \in x\,..\,Len(history) : \\
\qquad\quad history[x].term = history[y].term \Rightarrow history[x].node = history[y].node
\end{array}
$$

$$
\begin{array}{l}
\mathit{Init} \triangleq \\
\quad \land\ \mathit{term} = 0 \\
\quad \land\ \mathit{master} = \mathit{Nil} \\
\quad \land\ \mathit{backups} = \langle\rangle \\
\quad \land\ \mathit{events} = [n \in \mathit{Nodes} \mapsto \langle\rangle] \\
\quad \land\ \mathit{masterships} = [n \in \mathit{Nodes} \mapsto [\mathit{term} \mapsto 0,\ \mathit{master} \mapsto \mathit{Nil},\ \mathit{backups} \mapsto \langle\rangle]] \\
\quad \land\ \mathit{streams} = [n \in \mathit{Nodes} \mapsto [\mathit{state} \mapsto \mathit{Closed},\ \mathit{term} \mapsto 0]] \\
\quad \land\ \mathit{requests} = [n \in \mathit{Nodes} \mapsto \langle\rangle] \\
\quad \land\ \mathit{responses} = [n \in \mathit{Nodes} \mapsto \langle\rangle] \\
\quad \land\ \mathit{lastTerm} = 0 \\
\quad \land\ \mathit{mastershipChanges} = 0 \\
\quad \land\ \mathit{streamChanges} = 0 \\
\quad \land\ \mathit{messageCount} = 0 \\
\quad \land\ \mathit{history} = \langle\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathit{Next} \triangleq \\
\quad \lor \exists\, n \in \mathit{Nodes} : \mathit{ConnectStream}(n) \\
\quad \lor \exists\, n \in \mathit{Nodes} : \mathit{CloseStream}(n) \\
\quad \lor \exists\, n \in \mathit{Nodes} : \mathit{JoinMastershipElection}(n) \\
\quad \lor \exists\, n \in \mathit{Nodes} : \mathit{LeaveMastershipElection}(n) \\
\quad \lor \exists\, n \in \mathit{Nodes} : \mathit{LearnMastership}(n) \\
\quad \lor \exists\, n \in \mathit{Nodes} : \mathit{SendWriteRequest}(n) \\
\quad \lor \exists\, n \in \mathit{Nodes} : \mathit{HandleWrite}(n) \\
\quad \lor \exists\, n \in \mathit{Nodes} : \mathit{ReceiveWriteResponse}(n)
\end{array}
$$

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

* Modification History
* Last modified *Wed Feb* 20 16:09:21 *PST* 2019 by *jordanhalterman*
* Created *Thu Feb* 14 11:33:03 *PST* 2019 by *jordanhalterman*