

---

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*

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

VARIABLE *isMaster*

The state of all streams and their requests and responses

VARIABLE *streams, requests, responses*

The current set of elections for the device, the greatest of which is the current master

VARIABLE *elections*

The current set of terms for each open stream for the device

VARIABLE *terms*

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

$nodeVars \triangleq \langle events, masterships, isMaster \rangle$

Stream related variables

$streamVars \triangleq \langle streams, streamChanges \rangle$

Message related variables

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

Device related variables

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

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'

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

Indicates whether a request of type 't' is at the head of the queue for node 'n'

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

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

Indicates whether a response of type 't' is at the head of the queue for node 'n'  
 $HasResponse(n, t) \triangleq Len(responses[n]) > 0 \wedge responses[n][1].type = t$

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

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'.

$JoinMastershipElection(n) \triangleq$   
 $\wedge \vee \wedge master = Nil$   
 $\wedge term' = term + 1$   
 $\wedge master' = n$   
 $\wedge backups' = \langle \rangle$   
 $\wedge events' = [i \in Nodes \mapsto Append(events[i], [$   
 $term \mapsto term',$   
 $master \mapsto master',$   
 $backups \mapsto backups'])]$   
 $\vee \wedge master \neq Nil$

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

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.

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

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'

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

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.

$$\begin{aligned}
LearnMastership(n) \triangleq & \\
& \wedge Len(events[n]) > 0 \\
& \wedge LET \ e \triangleq events[n][1] \\
& \quad m \triangleq masterships[n] \\
& IN \\
& \quad \vee \wedge e.term > m.term \\
& \quad \quad \wedge masterships' = [masterships \text{ EXCEPT } ![n] = [ \\
& \quad \quad \quad term \mapsto e.term, \\
& \quad \quad \quad master \mapsto e.master, \\
& \quad \quad \quad backups \mapsto e.backups]] \\
& \quad \vee \wedge e.term = m.term \\
& \quad \quad \wedge masterships' = [masterships \text{ EXCEPT } ![n] = [ \\
& \quad \quad \quad term \mapsto e.term, \\
& \quad \quad \quad master \mapsto e.master, \\
& \quad \quad \quad backups \mapsto e.backups]] \\
& \wedge events' = [events \text{ EXCEPT } ![n] = Pop(events[n])] \\
& \wedge UNCHANGED \langle mastershipVars, isMaster, streamVars, messageVars, deviceVars \rangle
\end{aligned}$$

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.

$$\begin{aligned}
SendMasterArbitrationUpdate(n) \triangleq & \\
& \wedge IsStreamOpen(n) \\
& \wedge LET \ m \triangleq masterships[n] \\
& \quad s \triangleq streams[n] \\
& IN \\
& \quad \wedge m.term > 0 \\
& \quad \wedge s.term < m.term \\
& \quad \wedge \vee \wedge m.master = n \\
& \quad \quad \wedge SendRequest(n, [ \\
& \quad \quad \quad type \mapsto MasterArbitrationUpdate, \\
& \quad \quad \quad election\_id \mapsto MasterElectionId(m), \\
& \quad \quad \quad term \mapsto m.term]) \\
& \quad \vee \wedge m.master \neq n \\
& \quad \quad \wedge n \in Range(m.backups)
\end{aligned}$$

$$\begin{aligned}
& \wedge \text{SendRequest}(n, [ \\
& \quad \text{type} \quad \mapsto \text{MasterArbitrationUpdate}, \\
& \quad \text{election\_id} \mapsto \text{BackupElectionId}(n, m), \\
& \quad \text{term} \quad \mapsto m.\text{term}) \\
& \wedge \text{streams}' = [\text{streams} \text{ EXCEPT } ![n].\text{term} = m.\text{term}] \\
& \wedge \text{UNCHANGED } \langle \text{mastershipVars}, \text{events}, \text{masterships}, \text{isMaster}, \text{deviceVars}, \text{streamChanges}, \text{responses} \rangle
\end{aligned}$$

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.

$$\begin{aligned}
& \text{ReceiveMasterArbitrationUpdate}(n) \triangleq \\
& \quad \wedge \text{IsStreamOpen}(n) \\
& \quad \wedge \text{HasResponse}(n, \text{MasterArbitrationUpdate}) \\
& \quad \wedge \text{LET } r \triangleq \text{NextResponse}(n) \\
& \quad \quad m \triangleq \text{masterships}[n] \\
& \quad \quad s \triangleq \text{streams}[n] \\
& \quad \text{IN} \\
& \quad \quad \vee \wedge r.\text{status} = \text{Ok} \\
& \quad \quad \quad \wedge m.\text{master} = n \\
& \quad \quad \quad \wedge m.\text{term} = \text{MasterTerm}(r) \\
& \quad \quad \quad \wedge s.\text{term} = m.\text{term} \\
& \quad \quad \quad \wedge \text{isMaster}' = [\text{isMaster} \text{ EXCEPT } ![n] = \text{TRUE}] \\
& \quad \quad \vee \wedge \vee r.\text{status} \neq \text{Ok} \\
& \quad \quad \quad \vee m.\text{master} \neq n \\
& \quad \quad \quad \vee s.\text{term} \neq m.\text{term} \\
& \quad \quad \quad \vee m.\text{term} \neq \text{MasterTerm}(r) \\
& \quad \quad \quad \wedge \text{isMaster}' = [\text{isMaster} \text{ EXCEPT } ![n] = \text{FALSE}] \\
& \quad \wedge \text{DiscardResponse}(n) \\
& \quad \wedge \text{UNCHANGED } \langle \text{events}, \text{masterships}, \text{mastershipVars}, \text{deviceVars}, \text{streamVars}, \text{requests}, \text{messageCount} \rangle
\end{aligned}$$

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.

$$\begin{aligned}
& \text{SendWriteRequest}(n) \triangleq \\
& \quad \wedge \text{IsStreamOpen}(n) \\
& \quad \wedge \text{LET } m \triangleq \text{masterships}[n] \\
& \quad \text{IN} \\
& \quad \quad \wedge m.\text{term} > 0 \\
& \quad \quad \wedge m.\text{master} = n
\end{aligned}$$

$$\begin{aligned}
& \wedge isMaster[n] \\
& \wedge SendRequest(n, [ \\
& \quad type \mapsto WriteRequest, \\
& \quad election\_id \mapsto MasterElectionId(m), \\
& \quad term \mapsto m.term]) \\
& \wedge UNCHANGED \langle mastershipVars, nodeVars, deviceVars, streamVars, responses \rangle
\end{aligned}$$

Node 'n' receives a write response from the device

$$\begin{aligned}
ReceiveWriteResponse(n) & \triangleq \\
& \wedge IsStreamOpen(n) \\
& \wedge HasResponse(n, WriteResponse) \\
& \wedge LET m \triangleq NextResponse(n) \\
& IN \\
& \quad \vee m.status = Ok \\
& \quad \vee m.status = PermissionDenied \\
& \wedge DiscardResponse(n) \\
& \wedge UNCHANGED \langle mastershipVars, nodeVars, deviceVars, streamVars, requests, messageCount \rangle
\end{aligned}$$


---

This section models a *P4* Runtime device. For the purposes of this spec, the device has two functions: determine a master controller node and accept writes. Mastership is determined through *MasterArbitrationUpdates* sent by the controller nodes. The 'election\_id's provided by controller nodes are stored in 'elections', and the master is computed as the node with the highest 'election\_id' at any given time. The device will only allow writes from the current master node.

Returns the highest election *ID* for the given elections

$$DeviceElectionId(e) \triangleq Max(Range(e))$$

Returns the master for the given elections

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

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.

$$\begin{aligned}
ConnectStream(n) & \triangleq \\
& \wedge IsStreamClosed(n) \\
& \wedge streams' = [streams EXCEPT ![n].state = Open] \\
& \wedge streamChanges' = streamChanges + 1 \\
& \wedge UNCHANGED \langle mastershipVars, nodeVars, deviceVars, messageVars \rangle
\end{aligned}$$

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*, any 'election\_id' provided by node 'n' is forgotten, and the 'requests' and 'responses' queues for the node are cleared. Additionally, if the stream belonged to the master node, a new master is elected and a *MasterArbitrationUpdate* is sent on the streams that remain in the *Open* state. The *MasterArbitrationUpdate* will be sent to the new master with a 'status' of *Ok* and to all slaves with a 'status' of *AlreadyExists*.

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

The device receives and responds to a *MasterArbitrationUpdate* from node 'n'

If the 'election\_id' is already present in the 'elections' and does not already belong to node 'n', the stream is *Closed* and 'requests' and 'responses' are cleared for the node. If the 'election\_id' is not known to the device, it's added to the 'elections' state. If the change results in a new master being elected by the device, a *MasterArbitrationUpdate* is sent on all *Open* streams. If the change does not result in a new master being elected by the device, node 'n' is returned a

*MasterArbitrationUpdate*. The device master will always receive a *MasterArbitrationUpdate* response with 'status' of *Ok*, and slaves will always receive a 'status' of *AlreadyExists*.



$$\begin{aligned}
& \text{HandleMasterArbitrationUpdate}(n) \triangleq \\
& \quad \wedge \text{IsStreamOpen}(n) \\
& \quad \wedge \text{HasRequest}(n, \text{MasterArbitrationUpdate}) \\
& \quad \wedge \text{LET } m \triangleq \text{NextRequest}(n) \\
& \quad \text{IN} \\
& \quad \vee \wedge m.\text{election\_id} \in \text{Range}(\text{elections}) \\
& \quad \quad \wedge \text{elections}[n] \neq m.\text{election\_id} \\
& \quad \quad \wedge \text{streams}' = [\text{streams} \text{ EXCEPT } ![n] = [\text{state} \mapsto \text{Closed}, \text{term} \mapsto 0]] \\
& \quad \quad \wedge \text{requests}' = [\text{requests} \text{ EXCEPT } ![n] = \langle \rangle] \\
& \quad \quad \wedge \text{responses}' = [\text{responses} \text{ EXCEPT } ![n] = \langle \rangle] \\
& \quad \quad \wedge \text{UNCHANGED } \langle \text{deviceVars}, \text{streamChanges}, \text{messageCount} \rangle \\
& \quad \vee \wedge m.\text{election\_id} \notin \text{Range}(\text{elections}) \\
& \quad \quad \wedge \text{elections}' = [\text{elections} \text{ EXCEPT } ![n] = m.\text{election\_id}] \\
& \quad \quad \wedge \text{terms}' = [\text{terms} \text{ EXCEPT } ![n] = m.\text{term}] \\
& \quad \quad \wedge \text{LET } \text{oldMaster} \triangleq \text{DeviceMaster}(\text{elections}) \\
& \quad \quad \quad \text{newMaster} \triangleq \text{DeviceMaster}(\text{elections}') \\
& \quad \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 } \text{streams}[i].\text{state} = \text{Open} \text{ THEN} \\
& \quad \quad \quad \quad \quad \text{IF } i = \text{newMaster} \text{ THEN} \\
& \quad \quad \quad \quad \quad \quad \text{Append}(\text{responses}[i], [ \\
& \quad \quad \quad \quad \quad \quad \quad \text{type} \quad \quad \mapsto \text{MasterArbitrationUpdate}, \\
& \quad \quad \quad \quad \quad \quad \quad \text{status} \quad \mapsto \text{Ok}, \\
& \quad \quad \quad \quad \quad \quad \quad \text{election\_id} \mapsto \text{DeviceElectionId}(\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 \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{DeviceElectionId}(\text{elections}')) \\
& \quad \quad \quad \quad \text{ELSE} \\
& \quad \quad \quad \quad \quad \text{responses}[i]] \\
& \quad \quad \quad \quad \wedge \text{messageCount}' = \text{messageCount} + 1 \\
& \quad \quad \vee \wedge \text{oldMaster} = \text{newMaster} \\
& \quad \quad \quad \wedge \vee \wedge n = \text{newMaster} \\
& \quad \quad \quad \quad \wedge \text{SendResponse}(n, [ \\
& \quad \quad \quad \quad \quad \text{type} \quad \quad \mapsto \text{MasterArbitrationUpdate}, \\
& \quad \quad \quad \quad \quad \text{status} \quad \mapsto \text{Ok}, \\
& \quad \quad \quad \quad \quad \text{election\_id} \mapsto \text{DeviceElectionId}(\text{elections}')) \\
& \quad \quad \vee \wedge n \neq \text{newMaster} \\
& \quad \quad \quad \quad \wedge \text{SendResponse}(n, [ \\
& \quad \quad \quad \quad \quad \text{type} \quad \quad \mapsto \text{MasterArbitrationUpdate}, \\
& \quad \quad \quad \quad \quad \text{status} \quad \mapsto \text{AlreadyExists}, \\
& \quad \quad \quad \quad \quad \text{election\_id} \mapsto \text{DeviceElectionId}(\text{elections}')) \\
& \quad \quad \wedge \text{UNCHANGED } \langle \text{streamVars} \rangle
\end{aligned}$$

$\wedge \text{DiscardRequest}(n)$   
 $\wedge \text{UNCHANGED } \langle \text{mastershipVars}, \text{nodeVars}, \text{lastTerm}, \text{history} \rangle$

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

The *WriteRequest* is accepted if:

- \* The 'election\_id' for node 'n' matches the 'election\_id' for its stream
- \* Node 'n' is the current master for the device
- \* If node 'n' provided a 'term', the 'term' is greater than or equal to the highest term received by the device

When the *WriteRequest* is accepted, the 'lastTerm' is updated and the term of the node that sent the request is recorded for model checking. If the *WriteRequest* is rejected, a *PermissionDenied* response is returned.

$\text{HandleWrite}(n) \triangleq$   
 $\wedge \text{IsStreamOpen}(n)$   
 $\wedge \text{HasRequest}(n, \text{WriteRequest})$   
 $\wedge \text{LET } m \triangleq \text{NextRequest}(n)$   
 IN  
 $\vee \wedge \text{elections}[n] = m.\text{election\_id}$   
 $\wedge \text{DeviceMaster}(\text{elections}) = n$   
 $\wedge \vee \text{terms}[n] = 0$   
 $\vee \wedge \text{terms}[n] \geq \text{lastTerm}$   
 $\wedge \text{lastTerm}' = \text{terms}[n]$   
 $\wedge \text{history}' = \text{Append}(\text{history}, [\text{node} \mapsto n, \text{term} \mapsto m.\text{term}])$   
 $\wedge \text{SendResponse}(n, [$   
 $\quad \text{type} \mapsto \text{WriteResponse},$   
 $\quad \text{status} \mapsto \text{Ok}])$   
 $\vee \wedge \vee \text{elections}[n] \neq m.\text{election\_id}$   
 $\vee \text{DeviceMaster}(\text{elections}) \neq n$   
 $\vee \wedge \text{terms}[n] > 0$   
 $\wedge \text{terms}[n] < \text{lastTerm}$   
 $\wedge \text{SendResponse}(n, [$   
 $\quad \text{type} \mapsto \text{WriteResponse},$   
 $\quad \text{status} \mapsto \text{PermissionDenied}])$   
 $\wedge \text{UNCHANGED } \langle \text{lastTerm}, \text{history} \rangle$   
 $\wedge \text{DiscardRequest}(n)$   
 $\wedge \text{UNCHANGED } \langle \text{mastershipVars}, \text{nodeVars}, \text{elections}, \text{terms}, \text{streamVars} \rangle$

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.

$\text{TypeInvariant} \triangleq$   
 $\wedge \forall x \in 1 \dots \text{Len}(\text{history}) :$   
 $\quad \forall y \in x \dots \text{Len}(\text{history}) :$   
 $\quad \quad \text{history}[x].\text{term} \leq \text{history}[y].\text{term}$   
 $\wedge \forall x \in 1 \dots \text{Len}(\text{history}) :$

$$\forall y \in x \dots \text{Len}(\text{history}) : \\ \text{history}[x].\text{term} = \text{history}[y].\text{term} \Rightarrow \text{history}[x].\text{node} = \text{history}[y].\text{node}$$

---


$$\begin{aligned} \text{Init} &\triangleq \\ &\wedge \text{term} = 0 \\ &\wedge \text{master} = \text{Nil} \\ &\wedge \text{backups} = \langle \rangle \\ &\wedge \text{events} = [n \in \text{Nodes} \mapsto \langle \rangle] \\ &\wedge \text{masterships} = [n \in \text{Nodes} \mapsto [\text{term} \mapsto 0, \text{master} \mapsto \text{Nil}, \text{backups} \mapsto \langle \rangle]] \\ &\wedge \text{isMaster} = [n \in \text{Nodes} \mapsto \text{FALSE}] \\ &\wedge \text{streams} = [n \in \text{Nodes} \mapsto [\text{state} \mapsto \text{Closed}, \text{term} \mapsto 0]] \\ &\wedge \text{requests} = [n \in \text{Nodes} \mapsto \langle \rangle] \\ &\wedge \text{responses} = [n \in \text{Nodes} \mapsto \langle \rangle] \\ &\wedge \text{elections} = [n \in \text{Nodes} \mapsto 0] \\ &\wedge \text{terms} = [n \in \text{Nodes} \mapsto 0] \\ &\wedge \text{lastTerm} = 0 \\ &\wedge \text{mastershipChanges} = 0 \\ &\wedge \text{streamChanges} = 0 \\ &\wedge \text{messageCount} = 0 \\ &\wedge \text{history} = \langle \rangle \\ \\ \text{Next} &\triangleq \\ &\vee \exists n \in \text{Nodes} : \text{ConnectStream}(n) \\ &\vee \exists n \in \text{Nodes} : \text{CloseStream}(n) \\ &\vee \exists n \in \text{Nodes} : \text{JoinMastershipElection}(n) \\ &\vee \exists n \in \text{Nodes} : \text{LeaveMastershipElection}(n) \\ &\vee \exists n \in \text{Nodes} : \text{LearnMastership}(n) \\ &\vee \exists n \in \text{Nodes} : \text{SendMasterArbitrationUpdate}(n) \\ &\vee \exists n \in \text{Nodes} : \text{HandleMasterArbitrationUpdate}(n) \\ &\vee \exists n \in \text{Nodes} : \text{ReceiveMasterArbitrationUpdate}(n) \\ &\vee \exists n \in \text{Nodes} : \text{SendWriteRequest}(n) \\ &\vee \exists n \in \text{Nodes} : \text{HandleWrite}(n) \\ &\vee \exists n \in \text{Nodes} : \text{ReceiveWriteResponse}(n) \\ \\ \text{Spec} &\triangleq \text{Init} \wedge \Box[\text{Next}]_{\text{vars}} \end{aligned}$$


---

\\* Modification History  
\\* Last modified Wed Feb 20 17:53:13 PST 2019 by jordanhalterman  
\\* Created Thu Feb 14 11:33:03 PST 2019 by jordanhalterman