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

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

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

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'

$$\begin{array}{l} SendResponse(n,\ m) \triangleq \\ \quad \land\ responses' = [responses\ \text{EXCEPT}\ ![n] = Append(responses[n],\ m)] \\ \quad \land\ messageCount' = messageCount + 1 \end{array}$$

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'

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

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'.

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

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.

```
LeaveMastershipElection(n) ≜
    ∧ ∨ ∧ master = n
        ∧ ∨ ∧ Len(backups) > 0
            ∧ term' = term + 1
            ∧ master' = backups[1]
            ∧ backups' = Pop(backups)
            ∧ events' = [i    ∈ Nodes ↦ Append(events[i], [
                                          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, isMaster, streamVars, messageVars, deviceVars⟩
```

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) ≜ m.term + Cardinality(Nodes)
```

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

```
BackupElectionId(n, m) ≜ m.term + Cardinality(Nodes) − CHOOSE i ∈ DOMAIN m.backups : m.backups[i] =
```

Returns the mastership term for *MasterArbitrationUpdate* 'm'

```
MasterTerm(m) ≜ 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.

```
LearnMastership(n) ≜
    ∧ Len(events[n]) > 0
    ∧ LET e ≜ events[n][1]
          m ≜ masterships[n]
      IN
          ∨ ∧ e.term > m.term
            ∧ masterships' = [masterships EXCEPT ![n] = [
```

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\quad term \;\mapsto e.term,\\
\qquad\qquad\qquad\qquad\qquad\quad master \;\mapsto e.master,\\
\qquad\qquad\qquad\qquad\qquad\quad backups \mapsto e.backups,\\
\qquad\qquad\qquad\qquad\qquad\quad sent \;\mapsto \text{FALSE}]]\\
\qquad \lor \land e.term = m.term\\
\qquad\quad \land mastership s' = [mastership s \text{ EXCEPT } ![n] = [\\
\qquad\qquad\qquad\qquad\qquad\quad term \;\mapsto e.term,\\
\qquad\qquad\qquad\qquad\qquad\quad master \;\mapsto e.master,\\
\qquad\qquad\qquad\qquad\qquad\quad backups \mapsto e.backups,\\
\qquad\qquad\qquad\qquad\qquad\quad sent \;\mapsto m.sent]]\\
\land events' = [events \text{ EXCEPT } ![n] = Pop(events[n])]\\
\land \text{UNCHANGED } \langle mastership Vars,\ isMaster,\ stream Vars,\ message Vars,\ device Vars\rangle
\end{array}
$$

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 $election\_ids \leq 0$. 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.

$$
\begin{array}{l}
SendMasterArbitrationUpdate(n) \triangleq\\
\land streams[n] = Open\\
\land \text{LET } m \triangleq mastership s[n]\\
\quad \text{IN}\\
\qquad \land m.term > 0\\
\qquad \land \neg m.sent\\
\qquad \land \lor \land m.master = n\\
\qquad\qquad\quad \land SendRequest(n, [\\
\qquad\qquad\qquad\qquad type \;\mapsto MasterArbitrationUpdate,\\
\qquad\qquad\qquad\qquad election\_id \mapsto MasterElectionId(m),\\
\qquad\qquad\qquad\qquad term \;\mapsto m.term])\\
\qquad\qquad \lor \land m.master \neq n\\
\qquad\qquad\quad \land n \in Range(m.backups)\\
\qquad\qquad\quad \land SendRequest(n, [\\
\qquad\qquad\qquad\qquad type \;\mapsto MasterArbitrationUpdate,\\
\qquad\qquad\qquad\qquad election\_id \mapsto BackupElectionId(n,\ m),\\
\qquad\qquad\qquad\qquad term \;\mapsto m.term])\\
\land mastership s' = [mastership s \text{ EXCEPT } ![n].sent = \text{TRUE}]\\
\land \text{UNCHANGED } \langle mastership Vars,\ events,\ isMaster,\ device Vars,\ stream Vars,\ responses\rangle
\end{array}
$$

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.

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

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.

$$
\begin{array}{l}
SendWriteRequest(n) \triangleq \\
\quad \land\ streams[n] = Open \\
\quad \land\ \text{LET}\ m \triangleq mastership[n] \\
\qquad \text{IN} \\
\qquad\quad \land\ m.term > 0 \\
\qquad\quad \land\ m.master = n \\
\qquad\quad \land\ isMaster[n] \\
\qquad\quad \land\ SendRequest(n,\ [\\
\qquad\qquad\quad type \mapsto WriteRequest, \\
\qquad\qquad\quad election\_id \mapsto MasterElectionId(m), \\
\qquad\qquad\quad term \mapsto m.term]) \\
\quad \land\ \text{UNCHANGED}\ \langle mastershipVars,\ nodeVars,\ deviceVars,\ streamVars,\ responses \rangle
\end{array}
$$

Node 'n' receives a write response from the device

$$
\begin{array}{l}
ReceiveWriteResponse(n) \triangleq \\
\quad \land\ streams[n] = Open \\
\quad \land\ HasResponse(n,\ WriteResponse) \\
\quad \land\ \text{LET}\ m \triangleq NextResponse(n)
\end{array}
$$

IN

$\lor\ m.status = Ok$

$\lor\ m.status = PermissionDenied$

$\land\ DiscardResponse(n)$

$\land$ UNCHANGED $\langle mastershipVars,\ nodeVars,\ deviceVars,\ streamVars,\ requests,\ messageCount\rangle$

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

$DeviceMaster(e) \triangleq$

IF $Cardinality(\{i \in Range(e) : i > 0\}) > 0$ THEN

CHOOSE $n \in$ DOMAIN $e : e[n] = DeviceElectionId(e)$

ELSE

Nil

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.

$ConnectStream(n) \triangleq$

$\land\ streams[n] = Closed$

$\land\ streams' = [streams$ EXCEPT $![n] = Open]$

$\land\ streamChanges' = streamChanges + 1$

$\land$ UNCHANGED $\langle mastershipVars,\ nodeVars,\ deviceVars,\ messageVars\rangle$

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*, 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*.

$CloseStream(n) \triangleq$

$\land\ streams[n] = Open$

$\land\ elections' = [elections$ EXCEPT $![n] = 0]$

$\land\ streams' = [streams$ EXCEPT $![n] = Closed]$

$\land\ requests' = [requests$ EXCEPT $![n] = \langle\rangle]$

$\land$ LET $oldMaster \triangleq DeviceMaster(elections)$

$newMaster \triangleq DeviceMaster(elections')$

IN

$\lor\ \land\ oldMaster \neq newMaster$

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

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*.

```
HandleMasterArbitrationUpdate(n) ≜
    ∧ streams[n] = Open
    ∧ HasRequest(n, MasterArbitrationUpdate)
    ∧ LET m ≜ NextRequest(n)
      IN
         ∨ ∧ m.election_id ∈ Range(elections)
           ∧ elections[n] ≠ m.election_id
           ∧ streams′ = [streams EXCEPT ![n] = Closed]
           ∧ requests′ = [requests EXCEPT ![n] = ⟨⟩]
           ∧ responses′ = [responses EXCEPT ![n] = ⟨⟩]
           ∧ UNCHANGED ⟨deviceVars, streamChanges, messageCount⟩
         ∨ ∧ m.election_id ∉ Range(elections)
           ∧ elections′ = [elections EXCEPT ![n] = m.election_id]
           ∧ LET oldMaster ≜ DeviceMaster(elections)
                 newMaster ≜ DeviceMaster(elections′)
             IN
```

```
                    ∨ ∧ oldMaster ≠ newMaster
                      ∧ responses' = [i ∈ DOMAIN streams ↦
                                        IF streams[i] = Open THEN
                                            IF i = newMaster THEN
                                                Append(responses[i], [
                                                    type         ↦ MasterArbitrationUpdate,
                                                    status       ↦ Ok,
                                                    election_id ↦ DeviceElectionId(elections')])
                                             ELSE
                                                Append(responses[i], [
                                                    type         ↦ MasterArbitrationUpdate,
                                                    status       ↦ AlreadyExists,
                                                    election_id ↦ DeviceElectionId(elections')])
                                         ELSE
                                            responses[i]]
                      ∧ messageCount' = messageCount + 1
                    ∨ ∧ oldMaster = newMaster
                      ∧ ∨ ∧ n = newMaster
                          ∧ SendResponse(n, [
                                type         ↦ MasterArbitrationUpdate,
                                status       ↦ Ok,
                                election_id ↦ DeviceElectionId(elections')])
                        ∨ ∧ n ≠ newMaster
                          ∧ SendResponse(n, [
                                type         ↦ MasterArbitrationUpdate,
                                status       ↦ AlreadyExists,
                                election_id ↦ DeviceElectionId(elections')])
                ∧ UNCHANGED ⟨streamVars⟩
        ∧ DiscardRequest(n)
        ∧ UNCHANGED ⟨mastershipVars, nodeVars, history⟩
```

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

If the *WriteRequest* 'election_id' matches the 'election_id' recorded on the device for node 'n' and the node is the current master for the device, accept the write and record the term for model checking. Otherwise, return a 'PermissionDenied' response.

```
HandleWrite(n) ≜
    ∧ streams[n] = Open
    ∧ HasRequest(n, WriteRequest)
    ∧ LET m ≜ NextRequest(n)
      IN
          ∨ ∧ elections[n] = m.election_id
            ∧ DeviceMaster(elections) = n
            ∧ history' = Append(history, [node ↦ n, term ↦ m.term])
            ∧ SendResponse(n, [
                  type   ↦ WriteResponse,
```

$status \mapsto Ok])$
$\lor \land \lor elections[n] \neq m.election\_id$
$\lor DeviceMaster(elections) \neq n$
$\land SendResponse(n, [$
$type \mapsto WriteResponse,$
$status \mapsto PermissionDenied])$
$\land$ UNCHANGED $\langle history \rangle$
$\land DiscardRequest(n)$
$\land$ UNCHANGED $\langle mastershipVars,\ nodeVars,\ elections,\ streamVars \rangle$

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.

$TypeInvariant \triangleq \forall i \in$ DOMAIN $history : i = 1 \lor history[i-1].term \leq history[i].term$

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

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

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

* Modification History
* Last modified *Tue Feb* 19 17:59:56 *PST* 2019 by *jordanhalterman*
* Created *Thu Feb* 14 11:33:03 *PST* 2019 by *jordanhalterman*