MODULE *Config*

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

Indicates that a configuration change is waiting to be applied to the network
CONSTANT *Pending*

Indicates that a configuration change is being applied to the network
CONSTANT *Applying*

Indicates that a configuration change has been applied to the network
CONSTANT *Complete*

Indicates that a configuration change was successful
CONSTANT *Succeeded*

Indicates that a configuration change failed
CONSTANT *Failed*

Indicates a device is available
CONSTANT *Available*

Indicates a device is unavailable
CONSTANT *Unavailable*

Indicates that an error occurred when applying a change
CONSTANT *Error*

The set of all nodes
CONSTANT *Node*

The set of all devices
CONSTANT *Device*

An empty constant
CONSTANT *Nil*

Per-node election state
VARIABLE *leadership*

Per-node per-device election state
VARIABLE *mastership*

A sequence of network-wide configuration changes
Each change contains a record of 'changes' for each device
VARIABLE *networkChange*

A record of sequences of device configuration changes
Each sequence is a list of changes in the order in which they
are to be applied to the device

VARIABLE $deviceChange$

A record of sequences of pending configuration changes to each device.
VARIABLE $deviceQueue$

A record of device configurations derived from configuration chagnes pushed to each device.
VARIABLE $deviceConfig$

A record of device states - either *Available* or *Unavailable*
VARIABLE $deviceState$

A count of leader changes to serve as a state constraint
VARIABLE $electionCount$

A count of configuration changes to serve as a state constraint
VARIABLE $configCount$

A count of device availability changes to serve as a state constraint
VARIABLE $availabilityCount$

Node variables
$nodeVars \triangleq \langle leadership, mastership \rangle$

Configuration variables
$configVars \triangleq \langle networkChange, deviceChange \rangle$

Device variables
$deviceVars \triangleq \langle deviceQueue, deviceConfig, deviceState \rangle$

State constraint variables
$constraintVars \triangleq \langle electionCount, configCount, availabilityCount \rangle$

$vars \triangleq \langle leadership, mastership, networkChange, deviceChange, deviceConfig \rangle$

The invariant asserts that any configuration applied to a device implies that all prior configurations of the same device have been applied to all associated devices.

$$
\begin{array}{l}
TypeInvariant \triangleq \\
\quad \land \forall\, d \in \text{DOMAIN } deviceConfig : \\
\qquad deviceConfig[d] \neq 0 \Rightarrow \\
\qquad\quad Cardinality(\text{UNION } \{\{y \in \text{DOMAIN } deviceChange[x] : \\
\qquad\qquad\qquad\qquad \land\, deviceChange[x][y].network < deviceConfig[x] \\
\qquad\qquad\qquad\qquad \land\, deviceChange[x][y].status \neq Complete\} : \\
\qquad\qquad\qquad\qquad\quad x \in \text{DOMAIN } deviceChange\}) = 0
\end{array}
$$

This section models leader election for control loops and for devices. Leader election is modelled as a simple boolean indicating whether each node is the leader for the cluster and for each device. This model implies the ordering of leadership changes is irrelevant to the correctness of the spec.

Set the leader for node n to l

$$
\begin{array}{l}
SetNodeLeader(n,\ l) \triangleq \\
\quad \land\ leadership' = [leadership \text{ EXCEPT } ![n] = n = l] \\
\quad \land\ electionCount' = electionCount + 1 \\
\quad \land \text{ UNCHANGED } \langle mastership,\ configVars,\ deviceVars,\ configCount,\ availabilityCount \rangle
\end{array}
$$

Set the master for device d on node n to l

$$
\begin{array}{l}
SetDeviceMaster(n,\ d,\ l) \triangleq \\
\quad \land\ mastership' = [mastership \text{ EXCEPT } ![n] = [mastership[n] \text{ EXCEPT } ![d] = n = l]] \\
\quad \land\ electionCount' = electionCount + 1 \\
\quad \land \text{ UNCHANGED } \langle leadership,\ configVars,\ deviceVars,\ configCount,\ availabilityCount \rangle
\end{array}
$$

This section models the northbound *API* for the configuration service. The *API* exposes a single step to enqueue a configuration change. Rollback is not explicitly modelled as it can be implemented in an additional *Configure* step performing the inverse of the change being rolled back. When a configuration change is enqueued, it's simply added to network change for control loops to handle.

Enqueue network configuration change c

$$
\begin{array}{l}
Configure(c) \triangleq \\
\quad \land\ networkChange' = Append(networkChange, [\\
\qquad\qquad\qquad\qquad changes \mapsto c, \\
\qquad\qquad\qquad\qquad status \quad \mapsto Pending, \\
\qquad\qquad\qquad\qquad result \quad \mapsto Nil]) \\
\quad \land\ configCount' = configCount + 1 \\
\quad \land \text{ UNCHANGED } \langle nodeVars,\ deviceChange,\ deviceVars,\ electionCount,\ availabilityCount \rangle
\end{array}
$$

This section models a configuration change scheduler. The role of the scheduler is to determine when network changes can be applied and enqueue the relevant changes for application by changing their status from *Pending* to *Applying*. The scheduler supports concurrent application of non-overlapping configuration changes (changes that do not impact intersecting sets of devices) by comparing *Pending* changes with Applying changes.

Return the set of all network changes prior to the given change

$$
\begin{array}{l}
PriorNetworkChanges(c) \triangleq \\
\quad \{n \in \text{DOMAIN } networkChange : n < c\}
\end{array}
$$

Return the set of all completed device changes for network change c

$$
\begin{array}{l}
NetworkCompletedChanges(c) \triangleq \\
\quad \{d \in \text{DOMAIN } networkChange[c].changes : \\
\qquad \land\ Cardinality(\{x \in \text{DOMAIN } deviceChange[d] : deviceChange[d][x].network = c\}) \neq 0 \\
\qquad \land\ deviceChange[d][\text{CHOOSE } x \in \text{DOMAIN } deviceChange[d] \\
\qquad\qquad\qquad : deviceChange[d][x].network = c].status = Complete\}
\end{array}
$$

Return a boolean indicating whether all device changes are complete for the given network change

$NetworkChangesComplete(c) \triangleq$
$Cardinality(NetworkCompletedChanges(c)) = Cardinality(\text{DOMAIN } networkChange[c].changes)$

Return the set of all incomplete device changes prior to network change c
$PriorIncompleteDevices(c) \triangleq$
UNION $\{\text{DOMAIN } networkChange[n].changes :$
$n \in \{n \in PriorNetworkChanges(c) : \neg NetworkChangesComplete(n)\}\}$

Return the set of all devices configured by network change c
$NetworkChangeDevices(c) \triangleq \text{DOMAIN } networkChange[c].changes$

Return a boolean indicating whether network change c can be applied
A change can be applied if its devices do not intersect with past device
changes that have not been applied
$CanApply(c) \triangleq$
$Cardinality(NetworkChangeDevices(c) \cap PriorIncompleteDevices(c)) = 0$

Node n handles a network configuration change event c
$NetworkSchedulerNetworkChange(n,\ c) \triangleq$
$\land\ leadership[n] = \text{TRUE}$
$\land\ networkChange[c].status = Pending$
$\land\ CanApply(c)$
$\land\ networkChange' = [networkChange \text{ EXCEPT } ![c].status = Applying]$
$\land$ UNCHANGED $\langle nodeVars,\ deviceChange,\ deviceVars,\ constraintVars\rangle$

This section models the network-level change controller. The network control loop reacts to both network and device changes. The network controller runs on each node in the cluster, and the control loop can only be executed on a node that believes itself to be the leader. Note, however, that the model does not require a single leader.

When a network change is received:
- If the network change status is *Pending*
  - Add device changes for each configured device
- If the network change status is *Applying*
  - Update device change statuses to *Applying*

When a device change is received:
- If all device change statuses for the network are *Complete*
  - Mark the network change *Complete* with a *Succeeded* result if all device changes succeeded
  - Otherwise mark the network change *Complete* with a *Failed* result
- If all device changes failed due to *Unavailable* devices
  - Set the network change back to *Pending*
- If at least one device change succeeded but a subset failed
  - Fail the network change

Updates to network and device changes are atomic, and real-world implementations of the spec must provide for atomic updates for network and device changes as well. This can be done using either optimistic or pessimistic concurrency control.

Return a boolean indicating whether change c on device d already exists
$HasDeviceChange(d,\ c) \triangleq$

$Cardinality(\{x \in \text{DOMAIN } deviceChange[d] : deviceChange[d][x].network = c\}) \neq 0$

Return the index of the device change for network change c

$DeviceChange(d,\ c) \triangleq$
CHOOSE $x \in$ DOMAIN $deviceChange[d] : deviceChange[d][x].network = c$

Return a boolean indicating whether the device change for network change c has status s

$HasDeviceStatus(d,\ c,\ s) \triangleq$
$HasDeviceChange(d,\ c) \land deviceChange[d][DeviceChange(d,\ c)].status = s$

Add change c on device s

$AddDeviceChange(d,\ c) \triangleq$
IF $d \in$ DOMAIN $networkChange[c].changes \land \neg HasDeviceChange(d,\ c)$ THEN
$Append(deviceChange[d],\ [$
$network \mapsto c,$
$status \mapsto Pending,$
$value \mapsto networkChange[c].changes[d],$
$result \mapsto Nil,$
$reason \mapsto Nil])$
ELSE
$deviceChange[d]$

Change the status of change c on device s from *Pending* to *Applying*

$ApplyDeviceChange(d,\ c) \triangleq$
IF $d \in$ DOMAIN $networkChange[c].changes$ THEN
IF $HasDeviceChange(d,\ c)$ THEN
IF $HasDeviceStatus(d,\ c,\ Pending)$ THEN
$[deviceChange[d]$ EXCEPT $![DeviceChange(d,\ c)].status = Applying]$
ELSE
$deviceChange[d]$
ELSE
$Append(deviceChange[d],\ [$
$network \mapsto c,$
$status \mapsto Applying,$
$value \mapsto networkChange[c].changes[d],$
$result \mapsto Nil,$
$reason \mapsto Nil])$
ELSE
$deviceChange[d]$

Change the status of change c on device s to *Pending*

$PendDeviceChange(d,\ c) \triangleq$
IF $d \in$ DOMAIN $networkChange[c].changes$ THEN
$[deviceChange[d]$ EXCEPT $![DeviceChange(d,\ c)].status = Pending]$
ELSE
$deviceChange[d]$

```
 Return the set of all device changes for network change c
DeviceChanges(c) ≜
    {deviceChange[d][DeviceChange(d, c)] :
        d ∈ {d ∈ DOMAIN networkChange[c].changes : HasDeviceChange(d, c)}}

 Return a boolean indicating whether all device changes for network change c are complete
DeviceChangesComplete(c) ≜
    Cardinality({x ∈ DeviceChanges(c) : x.status = Complete}) = Cardinality(DeviceChanges(c))

 Return a boolean indicating whether all device changes for network change c were successful
DeviceChangesSucceeded(c) ≜
    Cardinality({x ∈ DeviceChanges(c) : x.result = Succeeded}) = Cardinality(DeviceChanges(c))

 Return a boolean indicating whether all device changes failed due to Unavailable devices
DeviceChangesUnavailable(c) ≜
    Cardinality({x ∈ DeviceChanges(c) : x.result = Failed ∧ x.reason = Unavailable}) =
        Cardinality(DeviceChanges(c))

 Node n handles a network configuration change c
NetworkControllerNetworkChange(n, c) ≜
    ∧ leadership[n] = TRUE
    ∧ LET change ≜ networkChange[c]IN
            ∨ ∧ change.status = Pending
              ∧ Cardinality({d ∈ DOMAIN networkChange[c].changes :
                      ¬HasDeviceStatus(d, c, Pending)}) > 0
              ∧ deviceChange′ = [d ∈ Device ↦ AddDeviceChange(d, c)]
            ∨ ∧ change.status = Applying
              ∧ Cardinality({d ∈ DOMAIN networkChange[c].changes :
                      ¬HasDeviceStatus(d, c, Applying)}) > 0
              ∧ deviceChange′ = [d ∈ Device ↦ ApplyDeviceChange(d, c)]
    ∧ UNCHANGED ⟨nodeVars, networkChange, deviceVars, constraintVars⟩

 Node n handles a device configuration change c
NetworkControllerDeviceChange(n, d, c) ≜
    ∧ leadership[n] = TRUE
    ∧ LET change ≜ deviceChange[d][c]
      IN
          ∧ change.status = Complete
          ∧ networkChange[change.network].status ≠ Complete
          ∧ ∨ ∧ DeviceChangesUnavailable(change.network)
              ∧ networkChange′ = [networkChange EXCEPT ![change.network] = [
                                      networkChange[change.network] EXCEPT
                                          !.status = Pending]]
            ∨ ∧ DeviceChangesComplete(change.network)
              ∧ DeviceChangesSucceeded(change.network)
              ∧ networkChange′ = [networkChange EXCEPT ![change.network] = [
```

```
                                networkChange[change.network] EXCEPT
                                    !.status = Complete, !.result = Succeeded]]
        ∨ ∧ DeviceChangesComplete(change.network)
          ∧ ¬DeviceChangesSucceeded(change.network)
          ∧ networkChange' = [networkChange EXCEPT ![change.network] = [
                                networkChange[change.network] EXCEPT
                                    !.status = Complete, !.result = Failed]]
    ∧ UNCHANGED ⟨nodeVars, deviceChange, deviceVars, constraintVars⟩
```

This section models the device-level change controller. The device control loop reacts to device changes and applies changes to devices. The device controller runs on each node in the cluster. A master is elected for each device, and the control loop can only be executed on the master for the device targeted by a change. Note, however, that the model does not require a single master per device. Multiple masters may exist for a device without violating safety properties.

When a device change is received: - If the node believes itself to be the master for the device and the change status
is *Applying*, apply the change
- Set the change status to *Complete*
- If the change was applied successfully, set the change result to *Succeeded*
- If the change failed, set the change result to *Failed*

Note: the above is modelled in two separate steps to allow the model checker to succeed and fail device changes.

Updates to network device changes are atomic, and real-world implementations of the spec must provide for atomic updates for network and device changes as well. This can be done using either optimistic or pessimistic concurrency control.

```
 Node n handles a device configuration change event c
DeviceControllerDeviceChange(n, d, c) ≜
    ∧ mastership[n][d] = TRUE
    ∧ LET change ≜ deviceChange[d][c]
      IN
          ∧ change.status = Applying
          ∧ Cardinality({i ∈ DOMAIN deviceQueue[n][d] : deviceQueue[n][d][i] = c}) = 0
          ∧ deviceQueue' = [deviceQueue EXCEPT ![n] = [
                                deviceQueue[n] EXCEPT ![d] = Append(deviceQueue[n][d], c)]]
    ∧ UNCHANGED ⟨nodeVars, configVars, deviceConfig, deviceState, constraintVars⟩

 Return a sequence with the head removed
Pop(q) ≜ SubSeq(q, 2, Len(q))

 Mark change c on device d succeeded
SucceedChange(n, d) ≜
    ∧ Len(deviceQueue[n][d]) > 0
    ∧ deviceState[d] = Available
    ∧ deviceChange' = [deviceChange EXCEPT ![d] = [
                        deviceChange[d] EXCEPT ![deviceQueue[n][d][1]] = [
                            deviceChange[d][deviceQueue[n][d][1]] EXCEPT
```

```
                                        !.status = Complete,
                                        !.result = Succeeded]]]
    ∧ deviceConfig' = [deviceConfig EXCEPT ![d] =
                            deviceChange[d][deviceQueue[n][d][1]].network]
    ∧ deviceQueue' = [deviceQueue EXCEPT ![n] = [
                            deviceQueue[n] EXCEPT ![d] = Pop(deviceQueue[n][d])]]
    ∧ UNCHANGED ⟨nodeVars, networkChange, deviceState, constraintVars⟩

Mark change c on device d failed
A change can be failed if the device rejects the change or the device is not reachable.
FailChange(n, d) ≜
    ∧ Len(deviceQueue[n][d]) > 0
    ∧ deviceChange' = [deviceChange EXCEPT ![d] = [
                        deviceChange[d] EXCEPT ![deviceQueue[n][d][1]] = [
                            deviceChange[d][deviceQueue[n][d][1]] EXCEPT
                                !.status = Complete,
                                !.result = Failed]]]
    ∧ deviceQueue' = [deviceQueue EXCEPT ![n] = [
                            deviceQueue[n] EXCEPT ![d] = Pop(deviceQueue[n][d])]]
    ∧ UNCHANGED ⟨nodeVars, networkChange, deviceConfig, deviceState, constraintVars⟩

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

This section models device states. Devices begin in the Unavailable state and can only be configured while in the Available state.

Set device d state to Available
ActivateDevice(d) ≜
    ∧ deviceState' = [deviceState EXCEPT ![d] = Available]
    ∧ availabilityCount' = availabilityCount + 1
    ∧ UNCHANGED ⟨nodeVars, configVars, deviceQueue, deviceConfig, electionCount, configCount⟩

Set device d state to Unavailable
DeactivateDevice(d) ≜
    ∧ deviceState' = [deviceState EXCEPT ![d] = Unavailable]
    ∧ availabilityCount' = availabilityCount + 1
    ∧ UNCHANGED ⟨nodeVars, configVars, deviceQueue, deviceConfig, electionCount, configCount⟩

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

Init and next state predicates
Init ≜
    ∧ leadership = [n ∈ Node ↦ FALSE]
    ∧ mastership = [n ∈ Node ↦ [d ∈ Device ↦ FALSE]]
    ∧ networkChange = ⟨⟩
    ∧ deviceChange = [d ∈ Device ↦ ⟨⟩]
    ∧ deviceQueue = [n ∈ Node ↦ [d ∈ Device ↦ ⟨⟩]]
    ∧ deviceConfig = [d ∈ Device ↦ 0]
    ∧ deviceState = [d ∈ Device ↦ Unavailable]
```

$\land\ electionCount = 0$
$\land\ configCount = 0$
$\land\ availabilityCount = 0$

$Next \triangleq$
$\lor \exists\, d \in \text{SUBSET } Device :$
$\quad Configure([x \in d \mapsto 1])$
$\lor \exists\, n \in Node :$
$\quad \exists\, l \in Node :$
$\quad\quad SetNodeLeader(n,\ l)$
$\lor \exists\, n \in Node :$
$\quad \exists\, d \in Device :$
$\quad\quad \exists\, l \in Node :$
$\quad\quad\quad SetDeviceMaster(n,\ d,\ l)$
$\lor \exists\, n \in Node :$
$\quad \exists\, c \in \text{DOMAIN } networkChange :$
$\quad\quad NetworkSchedulerNetworkChange(n,\ c)$
$\lor \exists\, n \in Node :$
$\quad \exists\, c \in \text{DOMAIN } networkChange :$
$\quad\quad NetworkControllerNetworkChange(n,\ c)$
$\lor \exists\, n \in Node :$
$\quad \exists\, d \in Device :$
$\quad\quad \exists\, c \in \text{DOMAIN } deviceChange[d] :$
$\quad\quad\quad NetworkControllerDeviceChange(n,\ d,\ c)$
$\lor \exists\, n \in Node :$
$\quad \exists\, d \in Device :$
$\quad\quad \exists\, c \in \text{DOMAIN } deviceChange[d] :$
$\quad\quad\quad DeviceControllerDeviceChange(n,\ d,\ c)$
$\lor \exists\, n \in Node :$
$\quad \exists\, d \in Device :$
$\quad\quad SucceedChange(n,\ d)$
$\lor \exists\, n \in Node :$
$\quad \exists\, d \in Device :$
$\quad\quad FailChange(n,\ d)$
$\lor \exists\, d \in Device :$
$\quad ActivateDevice(d)$
$\lor \exists\, d \in Device :$
$\quad DeactivateDevice(d)$

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

* Modification History
* Last modified *Mon Sep* 30 12:53:27 *PDT* 2019 by *jordanhalterman*
* Created *Fri Sep* 27 13:14:24 *PDT* 2019 by *jordanhalterman*