NetConf

Release master

Oct 07, 2019

Contents

CHAPTER 1

NETCONF Developer Guide

Note: Reading the NETCONF section in the User Guide is likely useful as it contains an overview of NETCONF in OpenDaylight and a how-to for spawning and configuring NETCONF connectors.

This chapter is recommended for application developers who want to interact with mounted NETCONF devices from their application code. It tries to demonstrate all the use cases from user guide with RESTCONF but now from the code level. One important difference would be the demonstration of NETCONF notifications and notification listeners. The notifications were not shown using RESTCONF because **RESTCONF does not support notifications from mounted NETCONF devices.**

Note: It may also be useful to read the generic OpenDaylight MD-SAL app development tutorial before diving into this chapter. This guide assumes awareness of basic OpenDaylight application development.

1.1 Sample app overview

All the examples presented here are implemented by a sample OpenDaylight application called **ncmount** in the `coretutorials` OpenDaylight project. It can be found on the github mirror of OpenDaylight's repositories:

- https://github.com/opendaylight/coretutorials/tree/stable/boron/ncmount

or checked out from the official OpenDaylight repository:

- https://git.opendaylight.org/gerrit/#/admin/projects/coretutorials

The application was built using the project startup maven archetype **and demonstrates how to:**

- preconfigure connectors to NETCONF devices
- retrieve MountPointService (registry of available mount points)
- listen and react to changing connection state of netconf-connector

- add custom device YANG models to the app and work with them
- read data from device in binding aware format (generated java APIs from provided YANG models)
- write data into device in binding aware format
- trigger and listen to NETCONF notifications in binding aware format

Detailed information about the structure of the application can be found at: https://wiki.opendaylight.org/view/Controller_Core_Functionality_Tutorials:Tutorials:Netconf_Mount

Note: The code in ncmount is fully **binding aware** (works with generated java APIs from provided YANG models). However it is also possible to perform the same operations in **binding independent** manner.

1.1.1 NcmountProvider

The NcmountProvider class (found in NcmountProvider.java) is the central point of the ncmount application and all the application logic is contained there. The following sections will detail its most interesting pieces.

Retrieve MountPointService

The MountPointService is a central registry of all available mount points in OpenDaylight. It is just another MD-SAL service and is available from the `session` attribute passed by `onSessionInitiated` callback:

```
@Override
public void onSessionInitiated(ProviderContext session) {
    LOG.info("NcmountProvider Session Initiated");

    // Get references to the data broker and mount service
    this.mountService = session.getSALService(MountPointService.class);

    ...

    }
}
```

Listen for connection state changes

It is important to know when a mount point appears, when it is fully connected and when it is disconnected or removed. The exact states of a mount point are:

- Connected
- Connecting
- Unable to connect

To receive this kind of information, an application has to register itself as a notification listener for the preconfigured netconf-topology subtree in MD-SAL's datastore. This can be performed in the `onSessionInitiated` callback as well:

```
@Override
public void onSessionInitiated(ProviderContext session) {
```

(continues on next page)

(continued from previous page)

```
    ...

    this.dataBroker = session.getSALService(DataBroker.class);

    // Register ourselves as the REST API RPC implementation
    this.rpcReg = session.addRpcImplementation(NcmountService.class, this);

    // Register ourselves as data change listener for changes on Netconf
    // nodes. Netconf nodes are accessed via "Netconf Topology" - a special
    // topology that is created by the system infrastructure. It contains
    // all Netconf nodes the Netconf connector knows about. NETCONF_TOPO_IID
    // is equivalent to the following URL:
    // .../restconf/operational/network-topology:network-topology/topology/topology-
↪netconf
    if (dataBroker != null) {
        this.dclReg = dataBroker.registerDataChangeListener(LogicalDatastoreType.
↪OPERATIONAL,
                NETCONF_TOPO_IID.child(Node.class),
                this,
                DataChangeScope.SUBTREE);
    }
}
```

The implementation of the callback from MD-SAL when the data change can be found in the `onDataChanged(AsyncDataChangeEvent<InstanceIdentifier<?>, DataObject> change)` callback of NcmountProvider class.

Reading data from the device

The first step when trying to interact with the device is to get the exact mount point instance (identified by an instance identifier) from the MountPointService:

```
@Override
public Future<RpcResult<ShowNodeOutput>> showNode(ShowNodeInput input) {
    LOG.info("showNode called, input {}", input);

    // Get the mount point for the specified node
    // Equivalent to '.../restconf/<config | operational>/opendaylight-
↪inventory:nodes/node/<node-name>/yang-ext:mount/'
    // Note that we can read both config and operational data from the same
    // mount point
    final Optional<MountPoint> xrNodeOptional = mountService.getMountPoint(NETCONF_
↪TOPO_IID
            .child(Node.class, new NodeKey(new NodeId(input.getNodeName()))));

    Preconditions.checkArgument(xrNodeOptional.isPresent(),
            "Unable to locate mountpoint: %s, not mounted yet or not configured",
            input.getNodeName());
    final MountPoint xrNode = xrNodeOptional.get();

    ....
}
```

Note: The triggering method in this case is called `showNode`. It is a YANG-defined RPC and NcmountProvider serves as an MD-SAL RPC implementation among other things. This means that `showNode` an be triggered using

RESTCONF.

The next step is to retrieve an instance of the `DataBroker` API from the mount point and start a read transaction:

```
@Override
public Future<RpcResult<ShowNodeOutput>> showNode(ShowNodeInput input) {

    ...

    // Get the DataBroker for the mounted node
    final DataBroker xrNodeBroker = xrNode.getService(DataBroker.class).get();
    // Start a new read only transaction that we will use to read data
    // from the device
    final ReadOnlyTransaction xrNodeReadTx = xrNodeBroker.newReadOnlyTransaction();

    ...
}
```

Finally, it is possible to perform the read operation:

```
@Override
public Future<RpcResult<ShowNodeOutput>> showNode(ShowNodeInput input) {

    ...

    InstanceIdentifier<InterfaceConfigurations> iid =
            InstanceIdentifier.create(InterfaceConfigurations.class);

    Optional<InterfaceConfigurations> ifConfig;
    try {
        // Read from a transaction is asynchronous, but a simple
        // get/checkedGet makes the call synchronous
        ifConfig = xrNodeReadTx.read(LogicalDatastoreType.CONFIGURATION, iid).
↪checkedGet();
    } catch (ReadFailedException e) {
        throw new IllegalStateException("Unexpected error reading data from " + input.
↪getNodeName(), e);
    }

    ...
}
```

The instance identifier is used here again to specify a subtree to read from the device. At this point application can process the data as it sees fit. The ncmount app transforms the data into its own format and returns it from `showNode`.

Note: More information can be found in the source code of ncmount sample app + on wiki: https://wiki.opendaylight.org/view/Controller_Core_Functionality_Tutorials:Tutorials:Netconf_Mount

CHAPTER 2

NETCONF User Guide

2.1 Overview

NETCONF is an XML-based protocol used for configuration and monitoring devices in the network. The base NETCONF protocol is described in RFC-6241.

NETCONF in OpenDaylight:.

OpenDaylight supports the NETCONF protocol as a northbound server as well as a southbound plugin. It also includes a set of test tools for simulating NETCONF devices and clients.

2.2 Southbound (netconf-connector)

The NETCONF southbound plugin is capable of connecting to remote NETCONF devices and exposing their configuration/operational datastores, RPCs and notifications as MD-SAL mount points. These mount points allow applications and remote users (over RESTCONF) to interact with the mounted devices.

In terms of RFCs, the connector supports:

- RFC-6241
- RFC-5277
- RFC-6022
- draft-ietf-netconf-yang-library-06

Netconf-connector is fully model-driven (utilizing the YANG modeling language) so in addition to the above RFCs, it supports any data/RPC/notifications described by a YANG model that is implemented by the device.

Tip: NETCONF southbound can be activated by installing `odl-netconf-connector-all` Karaf feature.

2.2.1 Netconf-connector configuration

There are 2 ways for configuring netconf-connector: NETCONF or RESTCONF. This guide focuses on using RESTCONF.

Important: There are 2 different endpoints related to RESTCONF protocols:

- `http://localhost:8181/restconf` is related to draft-bierman-netconf-restconf-02,
  can be activated by installing `odl-restconf-nb-bierman02` Karaf feature.
  This user guide uses this approach.
- `http://localhost:8181/rests` is related to RFC-8040,
  can be activated by installing `odl-restconf-nb-rfc8040` Karaf feature.

In case of RFC-8040 resources for configuration and operational datastores start `/rests/data/`,
e. g. GET http://localhost:8181/rests/data/network-topology:network-topology with response of both datastores. It's allowed to use query parameters to distinguish between them.
e. g. GET http://localhost:8181/rests/data/network-topology:network-topology?content=config for configuration datastore
and GET http://localhost:8181/rests/data/network-topology:network-topology?content=nonconfig for operational datastore.

Default configuration

The default configuration contains all the necessary dependencies (file: 01-netconf.xml) and a single instance of netconf-connector (file: 99-netconf-connector.xml) called **controller-config** which connects itself to the NETCONF northbound in OpenDaylight in a loopback fashion. The connector mounts the NETCONF server for config-subsystem in order to enable RESTCONF protocol for config-subsystem. This RESTCONF still goes via NETCONF, but using RESTCONF is much more user friendly than using NETCONF.

Spawning additional netconf-connectors while the controller is running

Preconditions:

1. OpenDaylight is running
2. In Karaf, you must have the netconf-connector installed (at the Karaf prompt, type: `feature:install odl-netconf-connector-all`); the loopback NETCONF mountpoint will be automatically configured and activated
3. Wait until log displays following entry: RemoteDevice{controller-config}: NETCONF connector initialized successfully

To configure a new netconf-connector you need to send following request to RESTCONF:

POST http://localhost:8181/restconf/config/network-topology:network-topology/topology/topology-netconf/node/controller-config/yang-ext:mount/config:modules

Headers:

- Accept application/xml
- Content-Type application/xml

```
<module xmlns="urn:opendaylight:params:xml:ns:yang:controller:config">
  <type xmlns:prefix=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">
↪prefix:sal-netconf-connector</type>
  <name>new-netconf-device</name>
  <address xmlns=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">127.0.0.1
↪</address>
  <port xmlns="urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf
↪">830</port>
  <username xmlns=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">admin</
↪username>
  <password xmlns=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">admin</
↪password>
  <tcp-only xmlns=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">false</
↪tcp-only>
  <event-executor xmlns=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">
    <type xmlns:prefix="urn:opendaylight:params:xml:ns:yang:controller:netty">
↪prefix:netty-event-executor</type>
    <name>global-event-executor</name>
  </event-executor>
  <binding-registry xmlns=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">
    <type xmlns:prefix="urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding
↪">prefix:binding-broker-osgi-registry</type>
    <name>binding-osgi-broker</name>
  </binding-registry>
  <dom-registry xmlns=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">
    <type xmlns:prefix="urn:opendaylight:params:xml:ns:yang:controller:md:sal:dom">
↪prefix:dom-broker-osgi-registry</type>
    <name>dom-broker</name>
  </dom-registry>
  <client-dispatcher xmlns=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">
    <type xmlns:prefix="urn:opendaylight:params:xml:ns:yang:controller:config:netconf
↪">prefix:netconf-client-dispatcher</type>
    <name>global-netconf-dispatcher</name>
  </client-dispatcher>
  <processing-executor xmlns=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">
    <type xmlns:prefix="urn:opendaylight:params:xml:ns:yang:controller:threadpool">
↪prefix:threadpool</type>
    <name>global-netconf-processing-executor</name>
  </processing-executor>
  <keepalive-executor xmlns=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">
    <type xmlns:prefix="urn:opendaylight:params:xml:ns:yang:controller:threadpool">
↪prefix:scheduled-threadpool</type>
    <name>global-netconf-ssh-scheduled-executor</name>
  </keepalive-executor>
</module>
```

This spawns a new netconf-connector which tries to connect to (or mount) a NETCONF device at 127.0.0.1 and port

830. You can check the configuration of config-subsystem's configuration datastore. The new netconf-connector will now be present there. Just invoke:

GET http://localhost:8181/restconf/config/network-topology:network-topology/topology/topology-netconf/node/controller-config/yang-ext:mount/config:modules

The response will contain the module for new-netconf-device.

Right after the new netconf-connector is created, it writes some useful metadata into the datastore of MD-SAL under the network-topology subtree. This metadata can be found at:

GET http://localhost:8181/restconf/operational/network-topology:network-topology/

Information about connection status, device capabilities, etc. can be found there.

Connecting to a device not supporting NETCONF monitoring

The netconf-connector in OpenDaylight relies on ietf-netconf-monitoring support when connecting to remote NETCONF device. The ietf-netconf-monitoring support allows netconf-connector to list and download all YANG schemas that are used by the device. NETCONF connector can only communicate with a device if it knows the set of used schemas (or at least a subset). However, some devices use YANG models internally but do not support NETCONF monitoring. Netconf-connector can also communicate with these devices, but you have to side load the necessary yang models into OpenDaylight's YANG model cache for netconf-connector. In general there are 2 situations you might encounter:

1. NETCONF device does not support ietf-netconf-monitoring but it does list all its YANG models as capabilities in HELLO message

This could be a device that internally uses only ietf-inet-types YANG model with revision 2010-09-24. In the HELLO message that is sent from this device there is this capability reported:

```
urn:ietf:params:xml:ns:yang:ietf-inet-types?module=ietf-inet-types&revision=2010-09-24
```

For such devices you only need to put the schema into folder cache/schema inside your Karaf distribution.

Important: The file with YANG schema for ietf-inet-types has to be called ietf-inet-types@2010-09-24.yang. It is the required naming format of the cache.

---

2. NETCONF device does not support ietf-netconf-monitoring and it does NOT list its YANG models as capabilities in HELLO message

Compared to device that lists its YANG models in HELLO message, in this case there would be no capability with ietf-inet-types in the HELLO message. This type of device basically provides no information about the YANG schemas it uses so its up to the user of OpenDaylight to properly configure netconf-connector for this device.

Netconf-connector has an optional configuration attribute called yang-module-capabilities and this attribute can contain a list of "YANG module based" capabilities. So by setting this configuration attribute, it is possible to override the "yang-module-based" capabilities reported in HELLO message of the device. To do this, we need to modify the configuration of netconf-connector by adding this XML (It needs to be added next to the address, port, username etc. configuration elements):

```
<yang-module-capabilities xmlns=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">
  <capability xmlns=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">
    urn:ietf:params:xml:ns:yang:ietf-inet-types?module=ietf-inet-types&amp;
↪revision=2010-09-24
```

(continues on next page)

(continued from previous page)

```
  </capability>
</yang-module-capabilities>
```

Remember to also put the YANG schemas into the cache folder.

Note: For putting multiple capabilities, you just need to replicate the capability xml element inside yang-module-capability element. Capability element is modeled as a leaf-list. With this configuration, we would make the remote device report usage of ietf-inet-types in the eyes of netconf-connector.

Reconfiguring Netconf-Connector While the Controller is Running

It is possible to change the configuration of a running module while the whole controller is running. This example will continue where the last left off and will change the configuration for the brand new netconf-connector after it was spawned. Using one RESTCONF request, we will change both username and password for the netconf-connector.

To update an existing netconf-connector you need to send following request to RESTCONF:

PUT http://localhost:8181/restconf/config/network-topology:network-topology/topology/topology-netconf/node/controller-config/yang-ext:mount/config:modules/module/odl-sal-netconf-connector-cfg:sal-netconf-connector/new-netconf-device

```
<module xmlns="urn:opendaylight:params:xml:ns:yang:controller:config">
  <type xmlns:prefix=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">
↪prefix:sal-netconf-connector</type>
  <name>new-netconf-device</name>
  <username xmlns=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">bob</
↪username>
  <password xmlns=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">passwd</
↪password>
  <tcp-only xmlns=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">false</
↪tcp-only>
  <event-executor xmlns=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">
    <type xmlns:prefix="urn:opendaylight:params:xml:ns:yang:controller:netty">
↪prefix:netty-event-executor</type>
    <name>global-event-executor</name>
  </event-executor>
  <binding-registry xmlns=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">
    <type xmlns:prefix="urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding
↪">prefix:binding-broker-osgi-registry</type>
    <name>binding-osgi-broker</name>
  </binding-registry>
  <dom-registry xmlns=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">
    <type xmlns:prefix="urn:opendaylight:params:xml:ns:yang:controller:md:sal:dom">
↪prefix:dom-broker-osgi-registry</type>
    <name>dom-broker</name>
  </dom-registry>
  <client-dispatcher xmlns=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">
```

(continues on next page)

(continued from previous page)

```
    <type xmlns:prefix="urn:opendaylight:params:xml:ns:yang:controller:config:netconf
↪">prefix:netconf-client-dispatcher</type>
    <name>global-netconf-dispatcher</name>
  </client-dispatcher>
  <processing-executor xmlns=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">
    <type xmlns:prefix="urn:opendaylight:params:xml:ns:yang:controller:threadpool">
↪prefix:threadpool</type>
    <name>global-netconf-processing-executor</name>
  </processing-executor>
  <keepalive-executor xmlns=
↪"urn:opendaylight:params:xml:ns:yang:controller:md:sal:connector:netconf">
    <type xmlns:prefix="urn:opendaylight:params:xml:ns:yang:controller:threadpool">
↪prefix:scheduled-threadpool</type>
    <name>global-netconf-ssh-scheduled-executor</name>
  </keepalive-executor>
</module>
```

Since a PUT is a replace operation, the whole configuration must be specified along with the new values for username and password. This should result in a 2xx response and the instance of netconf-connector called new-netconf-device will be reconfigured to use username bob and password passwd. New configuration can be verified by executing:

GET http://localhost:8181/restconf/config/network-topology:network-topology/topology/topology-netconf/node/controller-config/yang-ext:mount/config:modules/module/odl-sal-netconf-connector-cfg:sal-netconf-connector/new-netconf-device

With new configuration, the old connection will be closed and a new one established.

Destroying Netconf-Connector While the Controller is Running

Using RESTCONF one can also destroy an instance of a module. In case of netconf-connector, the module will be destroyed, NETCONF connection dropped and all resources will be cleaned. To do this, simply issue a request to following URL:

DELETE http://localhost:8181/restconf/config/network-topology:network-topology/topology/topology-netconf/node/controller-config/yang-ext:mount/config:modules/module/odl-sal-netconf-connector-cfg:sal-netconf-connector/new-netconf-device

The last element of the URL is the name of the instance and its predecessor is the type of that module (In our case the type is **sal-netconf-connector** and name **new-netconf-device**). The type and name are actually the keys of the module list.

2.2.2 Netconf-connector configuration with MD-SAL

It is also possible to configure new NETCONF connectors directly through MD-SAL with the usage of the network-topology model. You can configure new NETCONF connectors both through the NETCONF server for MD-SAL (port 2830) or RESTCONF. This guide focuses on RESTCONF.

Tip: To enable NETCONF connector configuration through MD-SAL install either the `odl-netconf-topology` or `odl-netconf-clustered-topology` feature. We will explain the difference between these features later.

Preconditions

1. OpenDaylight is running
2. In Karaf, you must have the `odl-netconf-topology` or `odl-netconf-clustered-topology` feature installed.
3. Feature `odl-restconf` must be installed
4. Wait until log displays following entry:

```
Successfully pushed configuration snapshot 02-netconf-topology.xml(odl-netconf-
↪topology,odl-netconf-topology)
```

or until

```
GET http://localhost:8181/restconf/operational/network-topology:network-topology/
↪topology/topology-netconf/
```

returns a non-empty response, for example:

```
<topology xmlns="urn:TBD:params:xml:ns:yang:network-topology">
  <topology-id>topology-netconf</topology-id>
</topology>
```

Spawning new NETCONF connectors

To create a new NETCONF connector you need to send the following request to RESTCONF:

```
PUT http://localhost:8181/restconf/config/network-topology:network-topology/topology/
↪topology-netconf/node/new-netconf-device
```

Headers:

- Accept: application/xml
- Content-Type: application/xml

Payload:

```
<node xmlns="urn:TBD:params:xml:ns:yang:network-topology">
  <node-id>new-netconf-device</node-id>
  <host xmlns="urn:opendaylight:netconf-node-topology">127.0.0.1</host>
  <port xmlns="urn:opendaylight:netconf-node-topology">17830</port>
  <username xmlns="urn:opendaylight:netconf-node-topology">admin</username>
  <password xmlns="urn:opendaylight:netconf-node-topology">admin</password>
  <tcp-only xmlns="urn:opendaylight:netconf-node-topology">false</tcp-only>
  <!-- non-mandatory fields with default values, you can safely remove these if you␣
↪do not wish to override any of these values-->
  <reconnect-on-changed-schema xmlns="urn:opendaylight:netconf-node-topology">false</
↪reconnect-on-changed-schema>
  <connection-timeout-millis xmlns="urn:opendaylight:netconf-node-topology">20000</
↪connection-timeout-millis>
  <max-connection-attempts xmlns="urn:opendaylight:netconf-node-topology">0</max-
↪connection-attempts>
  <between-attempts-timeout-millis xmlns="urn:opendaylight:netconf-node-topology">2000
↪</between-attempts-timeout-millis>
  <sleep-factor xmlns="urn:opendaylight:netconf-node-topology">1.5</sleep-factor>
```

(continues on next page)

(continued from previous page)

```
  <!-- keepalive-delay set to 0 turns off keepalives-->
  <keepalive-delay xmlns="urn:opendaylight:netconf-node-topology">120</keepalive-
↪delay>
</node>
```

Note that the device name in <node-id> element must match the last element of the restconf URL.

Reconfiguring an existing connector

The steps to reconfigure an existing connector are exactly the same as when spawning a new connector. The old connection will be disconnected and a new connector with the new configuration will be created.

Deleting an existing connector

To remove an already configured NETCONF connector you need to send the following:

```
DELETE http://localhost:8181/restconf/config/network-topology:network-topology/
↪topology/topology-netconf/node/new-netconf-device
```

Connecting to a device supporting only NETCONF 1.0

OpenDaylight is schema-based distribution and heavily depends on YANG models. However some legacy NETCONF devices are not schema-based and implement just RFC 4741. This type of device does not utilize YANG models internally and OpenDaylight does not know how to communicate with such devices, how to validate data, or what the semantics of data are.

NETCONF connector can communicate also with these devices, but the trade-offs are worsened possibilities in utilization of NETCONF mountpoints. Using RESTCONF with such devices is not suported. Also communicating with schemaless devices from application code is slightly different.

To connect to schemaless device, there is a optional configuration option in netconf-node-topology model called schemaless. You have to set this option to true.

2.2.3 Clustered NETCONF connector

To spawn NETCONF connectors that are cluster-aware you need to install the `odl-netconf-clustered-topology` karaf feature.

> **Warning:** The `odl-netconf-topology` and `odl-netconf-clustered-topology` features are considered **INCOMPATIBLE**. They both manage the same space in the datastore and would issue conflicting writes if installed together.

Configuration of clustered NETCONF connectors works the same as the configuration through the topology model in the previous section.

When a new clustered connector is configured the configuration gets distributed among the member nodes and a NETCONF connector is spawned on each node. From these nodes a master is chosen which handles the schema download from the device and all the communication with the device. You will be able to read/write to/from the device from all slave nodes due to the proxy data brokers implemented.

You can use the `odl-netconf-clustered-topology` feature in a single node scenario as well but the code that uses akka will be used, so for a scenario where only a single node is used, `odl-netconf-topology` might be preferred.

2.2.4 Netconf-connector utilization

Once the connector is up and running, users can utilize the new mount point instance. By using RESTCONF or from their application code. This chapter deals with using RESTCONF and more information for app developers can be found in the developers guide or in the official tutorial application **ncmount** that can be found in the coretutorials project:

- https://github.com/opendaylight/coretutorials/tree/stable/beryllum/ncmount

Reading data from the device

Just invoke (no body needed):

GET http://localhost:8080/restconf/operational/network-topology:network-topology/topology/topology-netconf/node/new-netconf-device/yang-ext:mount/

This will return the entire content of operation datastore from the device. To view just the configuration datastore, change **operational** in this URL to **config**.

Writing configuration data to the device

In general, you cannot simply write any data you want to the device. The data have to conform to the YANG models implemented by the device. In this example we are adding a new interface-configuration to the mounted device (assuming the device supports Cisco-IOS-XR-ifmgr-cfg YANG model). In fact this request comes from the tutorial dedicated to the **ncmount** tutorial app.

POST http://localhost:8181/restconf/config/network-topology:network-topology/topology/topology-netconf/node/new-netconf-device/yang-ext:mount/Cisco-IOS-XR-ifmgr-cfg:interface-configurations

```
<interface-configuration xmlns="http://cisco.com/ns/yang/Cisco-IOS-XR-ifmgr-cfg">
    <active>act</active>
    <interface-name>mpls</interface-name>
    <description>Interface description</description>
    <bandwidth>32</bandwidth>
    <link-status></link-status>
</interface-configuration>
```

Should return 200 response code with no body.

Tip: This call is transformed into a couple of NETCONF RPCs. Resulting NETCONF RPCs that go directly to the device can be found in the OpenDaylight logs after invoking `log:set TRACE org.opendaylight.controller.sal.connect.netconf` in the Karaf shell. Seeing the NETCONF RPCs might help with debugging.

This request is very similar to the one where we spawned a new netconf device. That's because we used the loopback netconf-connector to write configuration data into config-subsystem datastore and config-subsystem picked it up from there.

Invoking custom RPC

Devices can implement any additional RPC and as long as it provides YANG models for it, it can be invoked from OpenDaylight. Following example shows how to invoke the get-schema RPC (get-schema is quite common among netconf devices). Invoke:

POST http://localhost:8181/restconf/operations/network-topology:network-topology/topology/topology-netconf/node/new-netconf-device/yang-ext:mount/ietf-netconf-monitoring:get-schema

```
<input xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
  <identifier>ietf-yang-types</identifier>
  <version>2013-07-15</version>
</input>
```

This call should fetch the source for ietf-yang-types YANG model from the mounted device.

2.2.5 Netconf-connector + Netopeer

Netopeer (an open-source NETCONF server) can be used for testing/exploring NETCONF southbound in OpenDaylight.

Netopeer installation

A Docker container with netopeer will be used in this guide. To install Docker and start the netopeer image perform following steps:

1. Install docker http://docs.docker.com/linux/step_one/
2. Start the netopeer image:

   ```
   docker run -rm -t -p 1831:830 dockeruser/netopeer
   ```

3. Verify netopeer is running by invoking (netopeer should send its HELLO message right away:

   ```
   ssh root@localhost -p 1831 -s netconf
   (password root)
   ```

Mounting netopeer NETCONF server

Preconditions:

- OpenDaylight is started with features `odl-restconf-all` and `odl-netconf-connector-all`.
- Netopeer is up and running in docker

Now just follow the chapter: *Spawning netconf-connector*. In the payload change the:

- name, e.g., to netopeer
- username/password to your system credentials
- ip to localhost
- port to 1831.

After netopeer is mounted successfully, its configuration can be read using RESTCONF by invoking:

GET http://localhost:8181/restconf/config/network-topology:network-topology/topology/topology-netconf/node/netopeer/yang-ext:mount/

2.3 Northbound (NETCONF servers)

OpenDaylight provides 2 types of NETCONF servers:

- **NETCONF server for config-subsystem (listening by default on port 1830)**
  - Serves as a default interface for config-subsystem and allows users to spawn/reconfigure/destroy modules (or applications) in OpenDaylight
- **NETCONF server for MD-SAL (listening by default on port 2830)**
  - Serves as an alternative interface for MD-SAL (besides RESTCONF) and allows users to read/write data from MD-SAL's datastore and to invoke its rpcs (NETCONF notifications are not available in the Boron release of OpenDaylight)

Note: The reason for having 2 NETCONF servers is that config-subsystem and MD-SAL are 2 different components of OpenDaylight and require different approach for NETCONF message handling and data translation. These 2 components will probably merge in the future.

Note: Since Nitrogen release, there is performance regression in NETCONF servers accepting SSH connections. While opening a connection takes less than 10 seconds on Carbon, on Nitrogen time can increase up to 60 seconds. Please see https://bugs.opendaylight.org/show_bug.cgi?id=9020

2.3.1 NETCONF server for config-subsystem

This NETCONF server is the primary interface for config-subsystem. It allows the users to interact with config-subsystem in a standardized NETCONF manner.

In terms of RFCs, these are supported:

- RFC-6241
- RFC-5277
- RFC-6470
  - (partially, only the schema-change notification is available in Boron release)
- RFC-6022

For regular users it is recommended to use RESTCONF + the controller-config loopback mountpoint instead of using pure NETCONF. How to do that is spesific for each component/module/application in OpenDaylight and can be found in their dedicated user guides.

2.3.2 NETCONF server for MD-SAL

This NETCONF server is just a generic interface to MD-SAL in OpenDaylight. It uses the stadard MD-SAL APIs and serves as an alternative to RESTCONF. It is fully model driven and supports any data and rpcs that are supported by MD-SAL.

In terms of RFCs, these are supported:

- RFC-6241
- RFC-6022

- draft-ietf-netconf-yang-library-06

Notifications over NETCONF are not supported in the Boron release.

Tip: Install NETCONF northbound for MD-SAL by installing feature: `odl-netconf-mdsal` in karaf. Default binding port is **2830**.

Configuration

The default configuration can be found in file: *08-netconf-mdsal.xml*. The file contains the configuration for all necessary dependencies and a single SSH endpoint starting on port 2830. There is also a (by default disabled) TCP endpoint. It is possible to start multiple endpoints at the same time either in the initial configuration file or while OpenDaylight is running.

The credentials for SSH endpoint can also be configured here, the defaults are admin/admin. Credentials in the SSH endpoint are not yet managed by the centralized AAA component and have to be configured separately.

Verifying MD-SAL's NETCONF server

After the NETCONF server is available it can be examined by a command line ssh tool:

```
ssh admin@localhost -p 2830 -s netconf
```

The server will respond by sending its HELLO message and can be used as a regular NETCONF server from then on.

Mounting the MD-SAL's NETCONF server

To perform this operation, just spawn a new netconf-connector as described in *Spawning netconf-connector*. Just change the ip to "127.0.0.1" port to "2830" and its name to "controller-mdsal".

Now the MD-SAL's datastore can be read over RESTCONF via NETCONF by invoking:

GET http://localhost:8181/restconf/operational/network-topology:network-topology/topology/topology-netconf/node/controller-mdsal/yang-ext:mount

---

Note: This might not seem very useful, since MD-SAL can be accessed directly from RESTCONF or from Application code, but the same method can be used to mount and control other OpenDaylight instances by the "master OpenDaylight".

2.4 NETCONF testtool

NETCONF testtool is a set of standalone runnable jars that can:

- Simulate NETCONF devices (suitable for scale testing)
- Stress/Performance test NETCONF devices
- Stress/Performance test RESTCONF devices

These jars are part of OpenDaylight's controller project and are built from the NETCONF codebase in OpenDaylight.

Tip: Download testtool from OpenDaylight Nexus at: https://nexus.opendaylight.org/content/repositories/public/org/opendaylight/netconf/netconf-testtool/1.1.0-Boron/

Nexus contains 3 executable tools:

- executable.jar - device simulator
- stress.client.tar.gz - NETCONF stress/performance measuring tool
- perf-client.jar - RESTCONF stress/performance measuring tool

Tip: Each executable tool provides help. Just invoke `java -jar <name-of-the-tool.jar> --help`

2.4.1 NETCONF device simulator

NETCONF testtool (or NETCONF device simulator) is a tool that

- Simulates 1 or more NETCONF devices
- Is suitable for scale, performance or crud testing
- Uses core implementation of NETCONF server from OpenDaylight
- Generates configuration files for controller so that the OpenDaylight distribution (Karaf) can easily connect to all simulated devices
- Provides broad configuration options
- Can start a fully fledged MD-SAL datastore
- Supports notifications

Building testtool

1. Check out latest NETCONF repository from git
2. Move into the `opendaylight/netconf/tools/netconf-testtool/` folder
3. Build testtool using the `mvn clean install` command

Downloading testtool

Netconf-testtool is now part of default maven build profile for controller and can be also downloaded from nexus. The executable jar for testtool can be found at: nexus-artifacts

Running testtool

1. After successfully building or downloading, move into the `opendaylight/netconf/tools/netconf-testtool/target/` folder and there is file `netconf-testtool-1.1.0-SNAPSHOT-executable.jar` (or if downloaded from nexus just take that jar file)
2. Execute this file using, e.g.:

```
java -jar netconf-testtool-1.1.0-SNAPSHOT-executable.jar
```

This execution runs the testtool with default for all parameters and you should see this log output from the testtool :

```
10:31:08.206 [main] INFO  o.o.c.n.t.t.NetconfDeviceSimulator - Starting 1, SSH␣
↪simulated devices starting on port 17830
10:31:08.675 [main] INFO  o.o.c.n.t.t.NetconfDeviceSimulator - All simulated␣
↪devices started successfully from port 17830 to 17830
```

Default Parameters

The default parameters for testtool are:

- Use SSH
- Run 1 simulated device
- Device port is 17830
- YANG modules used by device are only: ietf-netconf-monitoring, ietf-yang-types, ietf-inet-types (these modules are required for device in order to support NETCONF monitoring and are included in the netconf-testtool)
- Connection timeout is set to 30 minutes (quite high, but when testing with 10000 devices it might take some time for all of them to fully establish a connection)
- Debug level is set to false
- No distribution is modified to connect automatically to the NETCONF testtool

Verifying testtool

To verify that the simulated device is up and running, we can try to connect to it using command line ssh tool. Execute this command to connect to the device:

```
ssh admin@localhost -p 17830 -s netconf
```

Just accept the server with yes (if required) and provide any password (testtool accepts all users with all passwords). You should see the hello message sent by simulated device.

Testtool help

```
usage: netconf testtool [-h] [--edit-content EDIT-CONTENT] [--async-requests {true,
↪false}] [--thread-amount THREAD-AMOUNT] [--throttle THROTTLE]
                        [--auth AUTH AUTH] [--controller-destination CONTROLLER-
↪DESTINATION] [--device-count DEVICES-COUNT]
                        [--devices-per-port DEVICES-PER-PORT] [--schemas-dir SCHEMAS-
↪DIR] [--notification-file NOTIFICATION-FILE]
                        [--initial-config-xml-file INITIAL-CONFIG-XML-FILE] [--
↪starting-port STARTING-PORT]
                        [--generate-config-connection-timeout GENERATE-CONFIG-
↪CONNECTION-TIMEOUT]
                        [--generate-config-address GENERATE-CONFIG-ADDRESS] [--
↪generate-configs-batch-size GENERATE-CONFIGS-BATCH-SIZE]
                        [--distribution-folder DISTRO-FOLDER] [--ssh {true,false}] [--
↪exi {true,false}] [--debug {true,false}]
```

(continues on next page)

(continued from previous page)

```
                        [--md-sal {true,false}] [--time-out TIME-OUT] [-ip IP] [--
↪thread-pool-size THREAD-POOL-SIZE] [--rpc-config RPC-CONFIG]

netconf testtool

named arguments:
  -h, --help            show this help message and exit
  --edit-content EDIT-CONTENT
  --async-requests {true,false}
  --thread-amount THREAD-AMOUNT
                        The number of threads to use for configuring devices.
  --throttle THROTTLE   Maximum amount of async requests that can be open at a time,␣
↪with mutltiple threads this gets divided among all threads
  --auth AUTH AUTH      Username and password for HTTP basic authentication in order␣
↪username password.
  --controller-destination CONTROLLER-DESTINATION
                        Ip address and port of controller. Must be in following ␣
↪format <ip>:<port> if available it will be used for spawning
                        netconf connectors via topology configuration ␣
↪as a part of URI. Example (http://<controller
                        destination>/restconf/config/network-topology:network-
↪topology/topology/topology-netconf/node/<node-id>)otherwise it will
                        just start simulated devices and skip the execution of PUT␣
↪requests
  --device-count DEVICES-COUNT
                        Number of simulated netconf devices to spin. This is the␣
↪number of actual ports open for the devices.
  --devices-per-port DEVICES-PER-PORT
                        Amount of config files generated per port to spoof more␣
↪devices than are actually running
  --schemas-dir SCHEMAS-DIR
                        Directory containing yang schemas to describe simulated␣
↪devices. Some schemas e.g. netconf monitoring and inet types are
                        included by default
  --notification-file NOTIFICATION-FILE
                        Xml file containing notifications that should be sent to␣
↪clients after create subscription is called
  --initial-config-xml-file INITIAL-CONFIG-XML-FILE
                        Xml file containing initial simulatted configuration to be␣
↪returned via get-config rpc
  --starting-port STARTING-PORT
                        First port for simulated device. Each other device will have␣
↪previous+1 port number
  --generate-config-connection-timeout GENERATE-CONFIG-CONNECTION-TIMEOUT
                        Timeout to be generated in initial config files
  --generate-config-address GENERATE-CONFIG-ADDRESS
                        Address to be placed in generated configs
  --generate-configs-batch-size GENERATE-CONFIGS-BATCH-SIZE
                        Number of connector configs per generated file
  --distribution-folder DISTRO-FOLDER
                        Directory where the karaf distribution for controller is␣
↪located
  --ssh {true,false}    Whether to use ssh for transport or just pure tcp
  --exi {true,false}    Whether to use exi to transport xml content
  --debug {true,false}  Whether to use debug log level instead of INFO
  --md-sal {true,false} Whether to use md-sal datastore instead of default simulated␣
↪datastore.
```

(continues on next page)

(continued from previous page)

```
  --time-out TIME-OUT    the maximum time in seconds for executing each PUT request
  -ip IP                 Ip address which will be used for creating a socket  address.
↪It  can  either  be a machine name, such as java.sun.com, or a
                         textual representation of its IP address.
  --thread-pool-size THREAD-POOL-SIZE
                         The number of threads to keep in the pool, when creating a␣
↪device simulator. Even if they are idle.
  --rpc-config RPC-CONFIG
                         Rpc config file. It can be used to define custom rpc ␣
↪behavior, or override the default one.Usable for testing buggy device
                         behavior.
```

Supported operations

Testtool default simple datastore supported operations:

get-schema returns YANG schemas loaded from user specified directory,

edit-config always returns OK and stores the XML from the input in a local variable available for get-config and get RPC. Every edit-config replaces the previous data,

commit always returns OK, but does not actually commit the data,

get-config returns local XML stored by edit-config,

get returns local XML stored by edit-config with netconf-state subtree, but also supports filtering.

(un)lock returns always OK with no lock guarantee

create-subscription returns always OK and after the operation is triggered, provided NETCONF notifications (if any) are fed to the client. No filtering or stream recognition is supported.

Note: when operation=”delete” is present in the payload for edit-config, it will wipe its local store to simulate the removal of data.

When using the MD-SAL datastore testtool behaves more like normal NETCONF server and is suitable for crud testing. create-subscription is not supported when testtool is running with the MD-SAL datastore.

Notification support

Testtool supports notifications via the –notification-file switch. To trigger the notification feed, create-subscription operation has to be invoked. The XML file provided should look like this example file:

```
<?xml version='1.0' encoding='UTF-8' standalone='yes'?>
<notifications>

<!-- Notifications are processed in the order they are defined in XML -->

<!-- Notification that is sent only once right after create-subscription is called -->
<notification>
    <!-- Content of each notification entry must contain the entire notification with␣
↪event time. Event time can be hardcoded, or generated by testtool if XXXX is set as␣
↪eventtime in this XML -->
    <content><![CDATA[
        <notification xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
            <eventTime>2011-01-04T12:30:46</eventTime>
            <random-notification xmlns="http://www.opendaylight.org/netconf/event:1.0
↪">
```

(continues on next page)

(continued from previous page)

```
                <random-content>single no delay</random-content>
            </random-notification>
        </notification>
    ]]></content>
</notification>

<!-- Repeated Notification that is sent 5 times with 2 second delay inbetween -->
<notification>
    <!-- Delay in seconds from previous notification -->
    <delay>2</delay>
    <!-- Number of times this notification should be repeated -->
    <times>5</times>
    <content><![CDATA[
        <notification xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
            <eventTime>XXXX</eventTime>
            <random-notification xmlns="http://www.opendaylight.org/netconf/event:1.0
↪">
                <random-content>scheduled 5 times 10 seconds each</random-content>
            </random-notification>
        </notification>
    ]]></content>
</notification>

<!-- Single notification that is sent only once right after the previous notification␣
↪-->
<notification>
    <delay>2</delay>
    <content><![CDATA[
        <notification xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
            <eventTime>XXXX</eventTime>
            <random-notification xmlns="http://www.opendaylight.org/netconf/event:1.0
↪">
                <random-content>single with delay</random-content>
            </random-notification>
        </notification>
    ]]></content>
</notification>

</notifications>
```

Connecting testtool with controller Karaf distribution

Auto connect to OpenDaylight

It is possible to make OpenDaylight auto connect to the simulated devices spawned by testtool (so user does not have to post a configuration for every NETCONF connector via RESTCONF). The testtool is able to modify the OpenDaylight distribution to auto connect to the simulated devices after feature `odl-netconf-connector-all` is installed. When running testtool, issue this command (just point the testool to the distribution:

```
java -jar netconf-testtool-1.1.0-SNAPSHOT-executable.jar --device-count 10 --
↪distribution-folder ~/distribution-karaf-0.4.0-SNAPSHOT/ --debug true
```

With the distribution-folder parameter, the testtool will modify the distribution to include configuration for netconf-connector to connect to all simulated devices. So there is no need to spawn netconf-connectors via RESTCONF.

Running testtool and OpenDaylight on different machines

The testtool binds by default to 0.0.0.0 so it should be accessible from remote machines. However you need to set the parameter "generate-config-address" (when using autoconnect) to the address of machine where testtool will be run so OpenDaylight can connect. The default value is localhost.

Executing operations via RESTCONF on a mounted simulated device

Simulated devices support basic RPCs for editing their config. This part shows how to edit data for simulated device via RESTCONF.

Test YANG schema

The controller and RESTCONF assume that the data that can be manipulated for mounted device is described by a YANG schema. For demonstration, we will define a simple YANG model:

```
module test {
    yang-version 1;
    namespace "urn:opendaylight:test";
    prefix "tt";

    revision "2014-10-17";


   container cont {

        leaf l {
            type string;
        }
   }
}
```

Save this schema in file called test@2014-10-17.yang and store it a directory called test-schemas/, e.g., your home folder.

Editing data for simulated device

- Start the device with following command:

```
java -jar netconf-testtool-1.1.0-SNAPSHOT-executable.jar --device-count 10 --
↪distribution-folder ~/distribution-karaf-0.4.0-SNAPSHOT/ --debug true --schemas-
↪dir ~/test-schemas/
```

- Start OpenDaylight
- Install odl-netconf-connector-all feature
- Install odl-restconf feature
- Check that you can see config data for simulated device by executing GET request to

```
http://localhost:8181/restconf/config/network-topology:network-topology/topology/
↪topology-netconf/node/17830-sim-device/yang-ext:mount/
```

- The data should be just and empty data container

- Now execute edit-config request by executing a POST request to:

```
http://localhost:8181/restconf/config/network-topology:network-topology/topology/
↪topology-netconf/node/17830-sim-device/yang-ext:mount
```

with headers:

```
Accept application/xml
Content-Type application/xml
```

and payload:

```
<cont xmlns="urn:opendaylight:test">
  <l>Content</l>
</cont>
```

- Check that you can see modified config data for simulated device by executing GET request to

```
http://localhost:8181/restconf/config/network-topology:network-topology/topology/
↪topology-netconf/node/17830-sim-device/yang-ext:mount/
```

- Check that you can see the same modified data in operational for simulated device by executing GET request to

```
http://localhost:8181/restconf/operational/network-topology:network-topology/
↪topology/topology-netconf/node/17830-sim-device/yang-ext:mount/
```

Warning: Data will be mirrored in operational datastore only when using the default simple datastore.

Testing User defined RPC

The NETCONF test-tool allows using custom RPC. Custom RPC needs to be defined in yang model provide to test-tool along with parameter `--schemas-dir`.

The input and output of the custom RPC should be provided with `--rpc-config` parameter as a path to the file containing definition of input and output. The format of the custom RPC file is xml as shown below.

Start the device with following command:

```
java -jar netconf/tools/netconf-testtool/target/netconf-testtool-1.7.0-SNAPSHOT-
↪executable.jar --schemas-dir ~/test-schemas/ --rpc-config ~/tmp/customrpc.xml --
↪debug=true
```

Example YANG model file:

```
module example-ops {
     namespace "urn:example-ops:reboot";
     prefix "ops";

    import ietf-yang-types {
    prefix "yang";
     }


     revision "2016-07-07" {
       description "Initial version.";
```

(continues on next page)

(continued from previous page)

```
      reference "example document.";
    }


    rpc reboot {
      description "Reboot operation.";
      input {
        leaf delay {
          type uint32;
          units "seconds";
          default 0;
          description
            "Delay in seconds.";
        }
        leaf message {
          type string;
          description
            "Log message.";
        }
      }
    }
  }
```

Example payload (RPC config file customrpc.xml):

```
<rpcs>
  <rpc>
    <input>
      <reboot xmlns="urn:example-ops:reboot">
        <delay>300</delay>
        <message>message</message>
      </reboot>
    </input>
    <output>
      <rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
        <ok/>
      </rpc-reply>
    </output>
  </rpc>
</rpcs>
```

Example of use:

```
POST http://localhost:8181/restconf/operations/network-topology:network-topology/
↪topology/topology-netconf/node/new-netconf-device/yang-ext:mount/example-ops:get-
↪reboot-info
```

If successful the command will return code 200.

Note: A working example of user defined RPC can be found in TestToolTest.java class of the tools[netconf-testtool] project.

Known problems

Slow creation of devices on virtual machines

When testtool seems to take unusually long time to create the devices use this flag when running it:

```
-Dorg.apache.sshd.registerBouncyCastle=false
```

Too many files open

When testtool or OpenDaylight starts to fail with TooManyFilesOpen exception, you need to increase the limit of open files in your OS. To find out the limit in linux execute:

```
ulimit -a
```

Example sufficient configuration in linux:

```
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 63338
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 500000
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 63338
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

To set these limits edit file: /etc/security/limits.conf, for example:

```
*         hard    nofile      500000
*         soft    nofile      500000
root      hard    nofile      500000
root      soft    nofile      500000
```

"Killed"

The testtool might end unexpectedly with a simple message: "Killed". This means that the OS killed the tool due to too much memory consumed or too many threads spawned. To find out the reason on linux you can use following command:

```
dmesg | egrep -i -B100 'killed process'
```

Also take a look at this file: /proc/sys/kernel/threads-max. It limits the number of threads spawned by a process. Sufficient (but probably much more than enough) value is, e.g., 126676

2.4.2 NETCONF stress/performance measuring tool

This is basically a NETCONF client that puts NETCONF servers under heavy load of NETCONF RPCs and measures the time until a configurable amount of them is processed.

2.4.3 RESTCONF stress-performance measuring tool

Very similar to NETCONF stress tool with the difference of using RESTCONF protocol instead of NETCONF.

2.5 YANGLIB remote repository

There are scenarios in NETCONF deployment, that require for a centralized YANG models repository. YANGLIB plugin provides such remote repository.

To start this plugin, you have to install odl-yanglib feature. Then you have to configure YANGLIB either through RESTCONF or NETCONF. We will show how to configure YANGLIB through RESTCONF.

2.5.1 YANGLIB configuration through RESTCONF

You have to specify what local YANG modules directory you want to provide. Then you have to specify address and port whre you want to provide YANG sources. For example, we want to serve yang sources from folder /sources on localhost:5000 adress. The configuration for this scenario will be as follows:

```
PUT  http://localhost:8181/restconf/config/network-topology:network-topology/topology/
↪topology-netconf/node/controller-config/yang-ext:mount/config:modules/module/
↪yanglib:yanglib/example
```

Headers:

- Accept: application/xml
- Content-Type: application/xml

Payload:

```
<module xmlns="urn:opendaylight:params:xml:ns:yang:controller:config">
  <name>example</name>
  <type xmlns:prefix="urn:opendaylight:params:xml:ns:yang:controller:yanglib:impl">
↪prefix:yanglib</type>
  <broker xmlns="urn:opendaylight:params:xml:ns:yang:controller:yanglib:impl">
    <type xmlns:prefix="urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding
↪">prefix:binding-broker-osgi-registry</type>
    <name>binding-osgi-broker</name>
  </broker>
  <cache-folder xmlns="urn:opendaylight:params:xml:ns:yang:controller:yanglib:impl">/
↪sources</cache-folder>
  <binding-addr xmlns="urn:opendaylight:params:xml:ns:yang:controller:yanglib:impl">
↪localhost</binding-addr>
  <binding-port xmlns="urn:opendaylight:params:xml:ns:yang:controller:yanglib:impl">
↪5000</binding-port>
</module>
```

This should result in a 2xx response and new YANGLIB instance should be created. This YANGLIB takes all YANG sources from /sources folder and for each generates URL in form:

```
http://localhost:5000/schemas/{modelName}/{revision}
```

On this URL will be hosted YANG source for particular module.

YANGLIB instance also write this URL along with source identifier to ietf-netconf-yang-library/modules-state/module list.

2.5.2 Netconf-connector with YANG library as fallback

There is an optional configuration in netconf-connector called yang-library. You can specify YANG library to be plugged as additional source provider into the mount's schema repository. Since YANGLIB plugin is advertising provided modules through yang-library model, we can use it in mount point's configuration as YANG library. To do this, we need to modify the configuration of netconf-connector by adding this XML

```
<yang-library xmlns="urn:opendaylight:netconf-node-topology">
  <yang-library-url xmlns="urn:opendaylight:netconf-node-topology">http://
↪localhost:8181/restconf/operational/ietf-yang-library:modules-state</yang-library-
↪url>
  <username xmlns="urn:opendaylight:netconf-node-topology">admin</username>
  <password xmlns="urn:opendaylight:netconf-node-topology">admin</password>
</yang-library>
```

This will register YANGLIB provided sources as a fallback schemas for particular mount point.

2.6 NETCONF Call Home

Important: The call home feature is experimental and will change in a future release. In particular, the Yang models will change to those specified in the RFC 8071

2.6.1 Call Home Installation

ODL Call-Home server is installed in Karaf by installing karaf feature `odl-netconf-callhome-ssh`. RESTCONF feature is recommended for configuring Call Home & testing its functionality.

```
feature:install odl-netconf-callhome-ssh
```

Note: In order to test Call Home functionality we recommend Netopeer. See Netopeer Call Home to learn how to enable call-home on Netopeer.

2.6.2 Northbound Call-Home API

The northbound Call Home API is used for administering the Call-Home Server. The following describes this configuration.

Global Configuration

Configuring global credentials

ODL Call-Home server allows user to configure global credentials, which will be used for devices which does not have device-specific credentials configured.

This is done by creating `/odl-netconf-callhome-server:netconf-callhome-server/global/credentials` with username and passwords specified.

Configuring global username & passwords to try

```
PUT
/restconf/config/odl-netconf-callhome-server:netconf-callhome-server/global/
↪credentials HTTP/1.1
Content-Type: application/json
Accept: application/json
```

```
{
  "credentials":
  {
    "username": "example",
    "passwords": [ "first-password-to-try", "second-password-to-try" ]
  }
}
```

Configuring to accept any ssh server key using global credentials

By default Netconf Call-Home Server accepts only incoming connections from allowed devices `/odl-netconf-callhome-server:netconf-callhome-server/allowed-devices`, if user desire to allow all incoming connections, it is possible to set `accept-all-ssh-keys` to `true` in `/odl-netconf-callhome-server:netconf-callhome-server/global`.

The name of this devices in `netconf-topology` will be in format `ip-address:port`. For naming devices see Device-Specific Configuration.

Allowing unknown devices to connect

This is a debug feature and should not be used in production. Besides being an obvious security issue, this also causes the Call-Home Server to drastically increase its output to the log.

```
POST
/restconf/config/odl-netconf-callhome-server:netconf-callhome-server/global HTTP/1.1
Content-Type: application/json
Accept: application/json
```

```
{
  "global": {
    "accept-all-ssh-keys": "true"
  }
}
```

Device-Specific Configuration

Allowing Device & Configuring Name

Netconf Call Home Server uses device provided SSH server key (host key) to identify device. The pairing of name and server key is configured in `/odl-netconf-callhome-server:netconf-callhome-server/allowed-devices`. This list is colloquially called a whitelist.

If the Call-Home Server finds the SSH host key in the whitelist, it continues to negotiate a NETCONF connection over an SSH session. If the SSH host key is not found, the connection between the Call Home server and the device is dropped immediately. In either case, the device that connects to the Call home server leaves a record of its presence in the operational store.

Example of configuring device

```
PUT
/restconf/config/odl-netconf-callhome-server:netconf-callhome-server/allowed-devices/
↪device/example HTTP/1.1
Content-Type: application/json
Accept: application/json
```

```
{
  "device": {
    "unique-id": "example",
    "ssh-host-key":
↪"AAAAB3NzaC1yc2EAAAADAQABAAABAQDHoH1jMjltOJnCt999uaSfc48ySutaD3ISJ9fSECe1Spdq9o9mxj0kBTTTq+2V8hPspu
↪rgJeoUewWwCAasRx9X4eTcRrJrwOQKzb5Fk+UKgQmenZ5uhLAefi2qXX/
↪agFCtZi99vw+jHXZStfHm9TZCAf2zi+HIBzoVksSNJD0VvPo66EAvLn5qKWQD4AdpQQbKqXRf5/
↪W8diPySbYdvOP2/7HFhDukW8yV/
↪7ZtcywFUIu3gdXsrzwMnTqnATSLPPuckoi0V2jd8dQvEcu1DY+rRqmqu0tEkFBurlRZDf1yhNzq5xWY3OXcjgDGN+RxwuWQK3cF
↪"
  }
}
```

Configuring Device with Device-specific Credentials

Call Home Server also allows to configure credentials per device basis, this is done by introducing `credentials` container into device-specific configuration. Format is same as in global credentials.

Configuring Device with Credentials

```
PUT
/restconf/config/odl-netconf-callhome-server:netconf-callhome-server/allowed-devices/
↪device/example HTTP/1.1
Content-Type: application/json
Accept: application/json
```

```
{
  "device": {
    "unique-id": "example",
    "credentials": {
      "username": "example",
      "passwords": [ "password" ]
    },
    "ssh-host-key":
↪"AAAAB3NzaC1yc2EAAAADAQABAAABAQDHoH1jMjltOJnCt999uaSfc48ySutaD3ISJ9fSECe1Spdq9o9mxj0kBTTTq+2V8hPspu
↪rgJeoUewWwCAasRx9X4eTcRrJrwOQKzb5Fk+UKgQmenZ5uhLAefi2qXX/
↪agFCtZi99vw+jHXZStfHm9TZCAf2zi+HIBzoVksSNJD0VvPo66EAvLn5qKWQD4AdpQQbKqXRf5/
↪W8diPySbYdvOP2/7HFhDukW8yV/
↪7ZtcywFUIu3gdXsrzwMnTqnATSLPPuckoi0V2jd8dQvEcu1DY+rRqmqu0tEkFBurlRZDf1yhNzq5xWY3OXcjgDGN+RxwuWQK3cF
↪"
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Operational Status

Once an entry is made into the config side of "allowed-devices", the Call-Home Server will populate an corresponding operational device that is the same as the config device but has an additional status. By default, this status is *DISCONNECTED*. Once a device calls home, this status will change to one of:

CONNECTED — The device is currently connected and the NETCONF mount is available for network management.

FAILED_AUTH_FAILURE — The last attempted connection was unsuccessful because the Call-Home Server was unable to provide the acceptable credentials of the device. The device is also disconnected and not available for network management.

FAILED_NOT_ALLOWED — The last attempted connection was unsuccessful because the device was not recognized as an acceptable device. The device is also disconnected and not available for network management.

FAILED — The last attempted connection was unsuccessful for a reason other than not allowed to connect or incorrect client credentials. The device is also disconnected and not available for network management.

DISCONNECTED — The device is currently disconnected.

Rogue Devices

Devices which are not on the whitelist might try to connect to the Call-Home Server. In these cases, the server will keep a record by instantiating an operational device. There will be no corresponding config device for these rogues. They can be identified readily because their device id, rather than being user-supplied, will be of the form "address:port". Note that if a device calls back multiple times, there will only be a single operatinal entry (even if the port changes); these devices are recognized by their unique host key.

2.6.3 Southbound Call-Home API

The Call-Home Server listens for incoming TCP connections and assumes that the other side of the connection is a device calling home via a NETCONF connection with SSH for management. The server uses port 6666 by default and this can be configured via a blueprint configuration file.

The device **must** initiate the connection and the server will not try to re-establish the connection in case of a drop. By requirement, the server cannot assume it has connectivity to the device due to NAT or firewalls among others.