NetconfX

Developer Guide
V2.1

Table of Contents

Figures

1 Introduction

The NetconfX package provides an implementation of the client-side of a NETCONF interface in the Java programming language. Using NetconfX, a developer can write Java applications that connect to NETCONF-enabled devices and configure them as well as receive asynchronous notifications from them. Effectively, the client-side of the following standards is supported:

- RFC 6261 (https://tools.ietf.org/html/rfc6261)
- RFC 5277 (https://tools.ietf.org/html/rfc5277)

This document describes what to do in order to get started with NetconfX quickly (without having to read and understand the companion architecture document). After reading this, you should be able to connect to a NETCONF device and perform some rudimentary operations (the equivalent of "Hello, World!").

In order to dig into more detail, please refer to the following (in increasing order of information overload):

- API Javadocs (bundled with the package)
- Source code (bundled with the package)

2 Distributions

There are two distinct distributions for NetconfX available on the download site:

- Binary – The binary distribution contains all the artifacts necessary to start using NetconfX in your project. It contains this document as well as the API Javadocs, which should allow a developer to use NetconfX in his application.
- Source – The source distribution the NetconfX project in its entirety – it comes packaged with all the source and build files needed to build NetconfX from scratch. A developer can use this package to make custom changes to source or use a debugger to solve issues at run-time.

2.1 Using the Binary Distribution

This is packaged as ZIP file. To unpack the distribution, extract the ZIP into an appropriate folder. When extracted, you will get the directory structure shown below:

- **3rdPARTY.html** – contains information about all the third-party software that is used by NetconfX. This file describes how to download these packages (in case you would like to get later versions than the ones bundled).
- **LICENSE** – describes the licensing terms for NetconfX.
- **README** – provides a brief one-page introduction to NetconfX.
- **lib** - contains all the Java packages that NetconfX itself depends on. These are:
  - centeredlogic-commons-1.2.jar
  - commons-logging.jar
  - commons-pool.jar
  - ganymed-ssh2-build210.jar
  - jdom.jar
  - junit-4.4.jar
  - uuid-3.1.jar
- **docs** – contains all the NetconfX documentation. This includes:
  - NetconfX-DeveloperGuide.pdf – This guide.
  - **api** – contains all the API JavaDocs for NetconfX.
- **dist** – contains the JAR (Java Archive) files that represent all the Java classes belonging to the NetconfX distribution. Includes:
  - netconfX-2.0.jar – Contains all the NetconfX client classes
  - netconfX_test-2.0.jar – Contains all the test classes used to validate NetconfX.

To use NetconfX, all you need to do in include the "*netconfX-1.0.jar*" file in your Java classpath, along with all the dependent JAR files (from the **lib** folder). You can pick and choose which of the dependent JAR files you need, based upon whether you are already using other (newer) versions of these in your project.

Note that, to run any of the test programs, you will need to include "netconfX_test-2.0.jar" in your Java classpath.

2.2 Using the Source distribution

This is packaged as ZIP file. To unpack the distribution, extract this into an appropriate folder. When extracted, you will get the directory structure shown below:

- **3rdPARTY.html** – contains information about all the third-party software that is used by NetconfX. This file describes how to download these packages (in case you would like to get later versions than the ones bundled).
- **LICENSE** – describes the licensing terms for NetconfX.
- **README** – provides a brief one-page introduction to NetconfX.
- **build.xml/build.properties –** Ant build files used to build the NetconfX product.
- **lib** - contains all the Java packages that NetconfX itself depends on. These are:
  - centeredlogic-commons-1.2.jar

  - commons-logging.jar
  - commons-pool.jar
  - ganymed-ssh2-build210.jar
  - jdom.jar
  - junit-4.4.jar
  - uuid-3.1.jar
- **docs** – contains all the NetconfX documentation. This includes:
  - QuickStartGuide.doc – This guide
- **src** – contains the source code used to build NetconfX. If you would like to build NetconfX from scratch, this folder contains a README file that provides instructions on how to build NetconfX using the "ant" build tool.
- **test** – contains source code used to test NetconfX. If you would like to look at sample code that uses NetconfX as examples (or as a starting point for your code), this folder provides a few good examples.

2.2.1 Building NetconfX with the source distribution

- Install the Java 1.6. SE development environment onto your system. NetconfX has been tested with JDK version 1.6, but lower versions will most likely work. Set up your path so that you can run "*javac*" (the Java compiler)
- Download Apache Ant onto your system. NetconfX has been tested with Ant 1.8, but lower versions should also work. Set up your path so that you can run Ant.
- Setup your environment variables that define where the NetconfX build artifacts end up:
  - export NETCONFX_BUILD=/NetconfX-build (*nix)
  - set NETCONFX_BUILD=C:\Temp\netconfx-build (Windows)
- Assuming you have extracted the NetconfX SDK into some appropriate folder, execute the ant tool:
  - ant bin_dist OR
  - ant -Ddebug=true bin_dist (if you want to debug the resultant code in a debugger)

  This will build all the sources to create the NetconfX distribution, which will be located at:

  $NETCONFX_BUILD/ netconfX-2.0.zip

  This ZIP file corresponds to the Binary distribution (as described in the previous section).

3 Developing with NetconfX

As a programmer, there are three tiers (or layers) at which you can use the NetconfX package. Each tier provides a different level of abstraction as shown below:

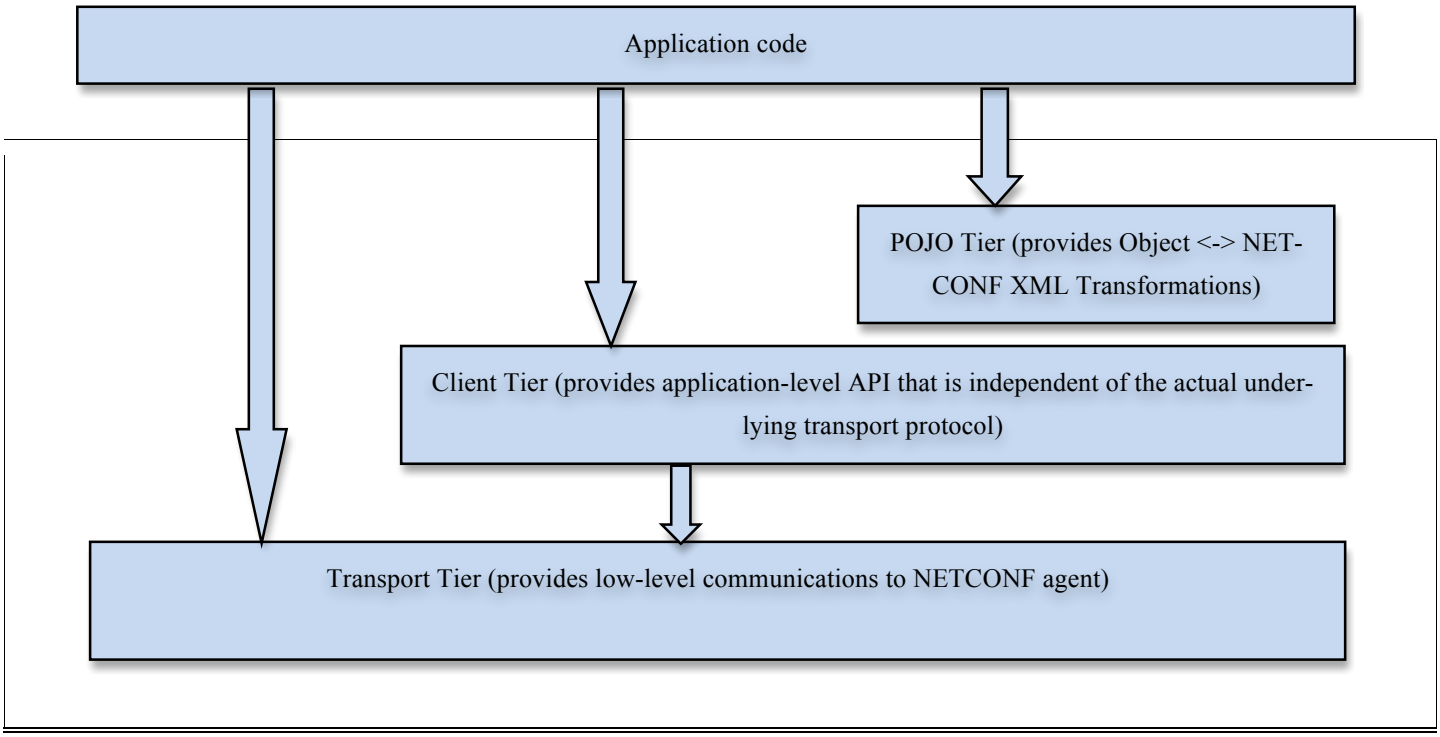

Figure 1 NetconfX Tiers

3.1 The Transport Tier

The basic NetconfX program described below operates at the transport layer (which is the lower-level API) used in the NetconfX stack. Primarily, the interface at this level is a XML exchange over SSH, without any NETCONF-specific semantics.

The sample code performs the following operations:

- Connect to the target device on the specified port., setting up an SSH connection to the device (with the appropriate security mechanism)
- Send out a NETCONF message (using the appropriate RPC) from a file.
- Get the response and display it.
- Terminate the connection.

3.1.1 Connecting to the target device

```
System.out.println("Setting up SSH transport client .. ");
SshTransportClient client = new SshTransportClient();
Properties properties = new Properties();
properties.put("host", <targetHostNameOrIPAddress>);
properties.put("socketTimeout", <timeoutValueInMilliseconds>);
properties.put("port", <port>));
properties.put("username", <sshUsername>);
properties.put("password", <sshPassword>);
// You could use a certificate with a passphrase instead of a password
client.setup(properties);
System.out.println("SSH transport client setup complete.");
```

3.1.2 Sending a NETCONF RPC request and displaying the response

If the above step succeeded, you have a valid SSH connection to the NETCONF subsystem on the target device. At this point, you are ready to send something to the device.

```
        // Read the request from the file
        Element request = XmlUtils.fromXmlFile(<filename>);
        System.out.println("Sending XML request: \n" + XmlUtils.toXmlString(request));
        // Send it out to the device synchronously, the response from the device
        // is what is returned
        Element response = client.send(request);
        System.out.println("Received XML response:\n"+XmlUtils.toXmlString(response));
```

3.1.3 Terminating the connection

Now that this is done, you are ready to disconnect from the device.

```
        client.shutdown();
```

For a richer set of example code which details how the transport layer is used, see the test class "*SshTransportClientTest.java*" in the test area. This class provides a number of test cases, including receiving notifications from a device.

3.2 The Client Tier

The Client layer is a transport-independent layer that provides NETCONF functionality at a higher level. Working at the Client level allows a programmer to use constructs that are defined in the NETCONF RFC, such as "*edit-config*", "*get-config*", etc. There are a number of features that the Client API supports:

- Support for all configuration RPC operations
- Support for registration of code to invoked upon receipt of notifications
- Support for "atomic" operations – i.e. guaranteed atomicity of multiple configuration operations.
- Support for probing device capabilities.

3.2.1 Setting up the Client

```
        System.out.println("Setting up NETCONF client");
        Properties properties = new Properties();
        properties.put("protocol", "ssh");    // Currently, only "ssh" is supported
        properties.put("host", <hostNameOrIPAddress>));
        properties.put("port", <port>));
        properties.put("username", <sshUsername>);
```

```
properties.put("password", <sshPassword>);
// You could use a certificate with a passphrase instead of a password
Client client = new Client();
client.setup(properties);
System.out.println("NETCONF client setup complete");
```

Once the Client has been set up successfully, the "hello" exchange with the device should be complete. You can now retrieve the capabilities presented by the device.

```
Capabilities caps = client.getDeviceCapabilities();
System.out.println(caps.toString());
// The Capabilities class contains the following information:
// 1. The set of device capability URLs
// 2. The set of notification streams supported by the device
```

3.2.2 Performing configuration operations

After set up, you can perform any of the configuration operations. Some examples are shown below:

```
Element filter = getStatusFilter();
// Do a NETCONF get() to get non-config data – argument passed in is
// the filter, return value is the "data" element of the RPC response.
Element data = client.get(filter, null);
// Do a NETCONF getConfig() – arguments passed in are the name of the
// configuration and the filter, return value is the "data" element
// of the RPC response.
filter = getConfigFilter();
data = client.getConfig("running", filter, null);
// Do a NETCONF editConfig() – arguments passed in are the name of the
// configuration and the edit XML
Element editXml = getEditConfigRequest();
client.editConfig("running", editXml, null, null, null, null);
// Do a NETCONF copyConfig() – arguments passed in are the name of the
// source and target configurations
client.copyConfig("running", "candidate", null);
// Do a NETCONF copyConfig() – argument passed in is the name of the
// target configuration
client.deleteConfig("candidate", null);
```

Note that, in general, if an error occurs in any of the API calls, a RuntimeException (or its subclass, NetconfException) is thrown. It is up to the application to catch it and handle it as appropriate.

3.2.3 Performing Transactional operations

The Client tier provides the concept of transactions – a transaction is a set of operations on a device delimited by a start/end transaction pair. When a transaction is started, the client returns a local transaction context containing a transaction ID, which is then passed in to all subsequent client calls. This ensures that all operations take place on the same underlying NETCONF session. An example is shown in the code below:

```
         try
         {
            NetconfLocalTransactionContext ctxt = (NetconfLocalTransactionContext)
m_client.startTransaction();
            // Get the transaction ID
            String xid = ctxt.getTransactionId();
            // Perform the first operation
            m_client.editConfig("candidate", getEditConfigXml("atomic1"), null, null,
null, xid);
            Thread.sleep(10000);
            // Perform the second operation
            // Note that if you read the running configuration here, you will NOT
            // see the changes made in the previous edit call
            m_client.editConfig("candidate", getEditConfigXml("atomic2"), null, null,
null, xid);
            // Note that nothing is committed to the device until you call commit
            m_client.commitTransaction(ctxt);
         }
         catch (final Exception ex)
         {
            System.out.println("Exception encountered: " + ex.getMessage());
         }
```

An atomic operation forces all the configuration operations to be performed using the same NETCONF session. A transaction involves the following steps:

- Bind a NETCONF session to the transaction ID when the transaction is started.
- Lock the “candidate” and “running” configurations, so that other sessions cannot make any changes.
- Perform all the edit-configuration operations.
- When the transaction is committed, commits the transaction using a NETCONF commit, which also unlocks the configurations.
- Unbinds the NETCONF session from the transaction ID.

If an unexpected error is encountered, the application should make sure the "candidate" and "running" configurations are unlocked and resources are released by calling:

```
            // Release resources
            m_client.rollbackTransaction(ctxt);
```

To see an example of atomic transaction, check out the test class "*ClientTest.java*" in the test area.

3.2.4 Notification Handling

NETCONF notifications are handled by registering a listener for a notification stream. A list of notification streams supported by the device is obtained by querying its capabilities as described above. Registration is done as shown below:

```
public class Listener implements NotificationListenerIf
{
    public void notify(Timestamp time, Element notification)
    {
       System.out.println("Notification:\n"+ XmlUtils.toXmlString(notification));
    }
}

       String streamName = <streamYouAreInterestedIn>;
   // If you want to replay messages from some specific time, pass a valid value.
   // If you want only new messages, you may pass in a NULL
   Timestamp replayFrom = <timeFromWhichReplayIsDesired>;
   NotificationListenerIf myListener = new Listener();
   client.startNotifications(streamName, replayFrom, listener);
```

When a notification arrives on the stream, the Client code invokes the *notify()* method of the listener, passing it the time-stamp of the notification (from the device) and the XML representing the information (or payload) of the notification. Any business logic based upon the notification can then be performed by the *notify()* method.

To stop receipt of notifications, use one of the *stop()* methods:

```
client.stopNotifications(<notificationStreamName>);
// Or, to stop notifications from ALL streams, call:
client.stopAllNotifications();
```

3.3 The POJO Tier

The POJO tier allows an application programmer to transform between POJOs (Plain Old Java Objects) and NETCONF XML. In order to perform this transformation, a mapping file that describes how Java objects map to NETCONF namespaces and XML documents is required. The bundle provides an example of one such mapping for the generic "toaster" device.

3.3.1 Mappings

A sample mapping file, which describes mappings from POJOs to NETCONF, is shown below. This file is called "mapping.xml" and corresponds to the NETCONF YANG file "toaster.yang", both of which can be found in the *com.centeredlogic.net.netconf.toaster* package in the "test" area. This package also contains the (generated) POJOs that are mapped by the mapping file.

```
<?xml version="1.0" encoding="UTF-8"?>
<netconfDataModel>
  <class name="com.centeredlogic.net.netconf.toaster.Toaster" xmlPath="" xmlCon-
tainer="" xmlTag="toaster" xmlNamespace="http://netconfcentral.org/ns/toaster"
keys="">
    <attribute name="toasterManufacturer" xmlContainer=""
xmlTag="toasterManufacturer" xmlNamespace="http://netconfcentral.org/ns/toaster" re-
adOnly="true" mandatory="true" many="false" />
    <attribute name="toasterModelNumber" xmlContainer="" xmlTag="toasterModelNumber"
xmlNamespace="http://netconfcentral.org/ns/toaster" readOnly="true" mandatory="true"
many="false" />
    <attribute name="toasterStatus" xmlContainer="" xmlTag="toasterStatus" xmlName-
space="http://netconfcentral.org/ns/toaster" readOnly="true" mandatory="true"
many="false" />
  </class>
  <class name="com.centeredlogic.net.netconf.toaster.ToastDone" xmlPath="" xmlCon-
tainer="" xmlTag="toastDone" xmlNamespace="http://netconfcentral.org/ns/toaster"
keys="">
    <attribute name="toastStatus" xmlContainer="" xmlTag="toastStatus" xmlName-
space="http://netconfcentral.org/ns/toaster" readOnly="false" mandatory="false"
many="false" />
```

```
  </class>
  <class name="com.centeredlogic.net.netconf.toaster.MakeToast" xmlPath="" xmlCon-
tainer="" xmlTag="make-toast" xmlNamespace="http://netconfcentral.org/ns/toaster"
keys="">
    <attribute name="toasterDoneness" xmlContainer="" xmlTag="toasterDoneness"
xmlNamespace="http://netconfcentral.org/ns/toaster" readOnly="false" manda-
tory="false" many="false" />
    <attribute name="toasterToastType" xmlContainer="" xmlTag="toasterToastType"
xmlNamespace="http://netconfcentral.org/ns/toaster" readOnly="false" manda-
tory="false" many="false" />
  </class>
  <class name="com.centeredlogic.net.netconf.toaster.CancelToast" xmlPath="" xmlCon-
tainer="" xmlTag="cancel-toast" xmlNamespace="http://netconfcentral.org/ns/toaster"
keys="" />
</netconfDataModel>
```

Using this mapping file (and the POJO tier) you can marshal and un-marshall NETCONF XML to the POJOs and back.

3.3.2 Setting up the mapping

To set up the mapping, you need to load up the data model from the XML mapping file as shown below. The data model holds the metadata representing the POJOs and their transformations.

```
        Element modelXml = XmlUtils.fromXmlFile(<mappingFileName>));
        DataModel dm = new DataModel();
        dm.fromXml(modelXml);
```

Once you have the data model, you can use the Filter and NetconfUtil classes to retrieve and send these modeled objects from/to the target device.

3.3.3 Object conversion to NETCONF

Assume you want to create an object on the device. Let us consider the example of the Toaster object from the above mapping file (although this example is not perfectly correct, since all the attributes of the Toaster object are read-only and hence cannot be set).

The first step is to generate the NETCONF XML that corresponds to the creation of a Toaster POJO, using the NetconfUtil class. This is done as shown below:

```
// Instantiate the object you want to create or update
Toaster t = new Toaster();
// Set its variables (assuming they are not read-only)
t.setManufacturer("BurntBread, Inc");
// etc. etc …
// Since the Toaster class does not have a parent configuration
// class, the path to the parent is NULL.
Path parentPath = null;
// Instantiate the NetconfUtil class, passing it the data model
//representing the mappings
NetconfUtil nu = new NetconfUtil(dm);
Element objectRoot = nu.toXml(path, null, o, EditOperation.Create);
```

At this point, you have the NETCONF XML that represents the POJO. This XML needs to be enveloped inside NETCONF RPC XML and sent to the target device using the Client tier (described in Section 3.2) as shown below:

```
client.editConfig(<storeName>, objectRoot, null, null, null, null);
```

The example code below performs an object update (this is very similar to object creation, except for the EditOperation type and passing in the old object):

```
// Assume the old Toaster is in "old" and the new one in "new"
Toaster old = … ;
Toaster new = … ;
// Modify its variables (assuming they are not read-only)
new.setManufacturer("NewManufacturer, Inc");
// etc. etc …
// Since the Toaster class does not have a parent configuration
// class, the path to the parent is NULL.
Path parentPath = null;
// We need the old object in case something needs to be "unset"
Element objectRoot = nu.toXml(path, old, new, EditOperation.Update);
client.editConfig(<storeName>, objectRoot, null, null, null, null);
```

3.3.4 Object retrieval via NETCONF

Object retrieval is performed by using the Filter class to define what is returned from the device. The Filter class provides three types of operations, detailed in the following sections.

3.3.4.1 Retrieving all children of a certain type, given a parent.

```
ClassMapping parentClass = dm.getClassMappingByName(<parentJavaClassName>);
ClassMapping childClass = dm.getClassMappingByName(<childJavaClassName>);
// The parent OID is an Xpath – e.g. /toaster[name="foo"]
String parentOid = <oidOfParent>;
Filter f = new Filter(parentClass, parentOid, childClass);
Element filter = f.getAsXml();
// At this point, you have the XML representing the filter
Element data = null;
if (storeName == null)
{
   // In case you are not doing a configuration get
   data = m_client.get(filter, null);
}
else
{
   // In case you are doing a configuration get
   data = m_client.getConfig(storeName, filter, null);
}
List<Element> returnedData = (List<Element>) data.getChildren();
if (returnedData == null || returnedData.size() == 0)
{
   // Nothing was returned; so return an empty list
   return new ArrayList<Object>();
}
// Instantiate the NetconfUtil class, passing it the data model
NetconfUtil nu = new NetconfUtil(dm);
return nu.fromXml(data, childClass.getJavaClassName());
```

3.3.4.2 Retrieving a specific object instance.

Retrieving a specific object instance is very similar to getting all the children of a parent; the primary change being that the NETCONF filter contains the ID of the instance we are interested in. This derives from the fact that an objects instance ID (or OID) is an Xpath expression which also contains the parent's OID. For example, if you have an object with an OID like:

/aaa/bbb/ccc[key="1"]/ddd[x="2"][y="3"]

its parents OID is

/aaa/bbb/ccc[key="1"]

```
ClassMapping childClass = dm.getClassMappingByName(<childJavaClassName>);
// The OID is an Xpath – e.g. /toaster[name="foo"]
String oid = <oidOfInstance>;
Filter f = new Filter(childClass, oid);
Element filter = f.getAsXml();
```

```
      // At this point, you have the XML representing the filter
      Element data = null;
      if (storeName == null)
      {
         // In case you are not doing a configuration get
         data = m_client.get(filter, null);
      }
      else
      {
         // In case you are doing a configuration get
         data = m_client.getConfig(storeName, filter, null);
      }
      List<Element> returnedData = (List<Element>) data.getChildren();
      if (returnedData == null || returnedData.size() == 0)
      {
         // Nothing was returned; so return NULL
         return null;
      }
      // Instantiate the NetconfUtil class, passing it the data model
      NetconfUtil nu = new NetconfUtil(dm);
      List<Object> vos = nu.fromXml(data, cmap.getJavaClassName());
      if (vos.size() > 1)
      {
         throw new Exception("Expected one object for" + type + "; got: " +
vos.size() + " objects instead.");
      }
      if (vos.size() == 0)
      {
         // No objects were returned; so return NULL
         return null;
      }
      return vos.get(0);
```

3.3.4.3 Getting the NETCONF XML that corresponds to deleting a specific instance.

Deleting an instance of an object involves generating a filter XML similar to the instance retrieval, except that we set the delete flag at the object level in the generated filter XML, so that the device gets rid of the object (and its descendants) in the tree.

```
      ClassMapping cmap = dm.getClassMappingByName(<objectJavaClassName>);
      String oid = <oidOfObjectToDelete>;
      // Instantiate the filter with delete flag set to true
      Filter f = new Filter(cmap, oid, true);
      Element filter = f.getAsXml();
      // At this point, we have the XML for the delete
      // Just call the NETCONF edit-config command to delete the object
```

```
client.editConfig(<storeName>, filter, null, null, null, null);
```

To see a much more complex example of the relationship between a YANG model and its corresponding POJO mapping XML, bundled with NetconfX is a pair of files under the "sample" directory:

- onf-config.yang
- onf-config-pojo-model.xml

These models represent configuration of devices supporting OpenFlow 1.1 (used for Software Defined Networking). For more information, please see https://www.opennetworking.org/about/onf-overview.

4 Distributed (or global) Transactions

It is possible to run a distributed transaction using NETCONF across multiple target devices, provided they support the "candidate configuration" and "confirmed commit" capabilities. This feature is useful for situations wherein you need to configure multiple devices as a single set, and you want **all** the configuration changes to take effect (or none at all).

NetconfX provides this functionality by leveraging the transactional operation methodology (described in section 3.2.4). It introduces the concept of a *GlobalTransaction* object, which manages a number of *TransactionBranch* objects, Each *TransactionBranch* is responsible for atomic operations on a device and manages the life-cycle of each transactional operation. A step-by-step guide of how to perform global transactions is given below.

4.1 Setup the Clients that communicate with the devices

```
ArrayList<Client> clients = new ArrayList<Client>();
for (int i=0; i<<numberOfClients>; i++)
{
   Properties properties = new Properties();
   properties.put("host", <hostname[i]>);
   properties.put("port", <port[i]>);
   properties.put("username",<username[i]>);
   properties.put("password",<password[i]>);
   Client client = new Client();
```

```
            client.setup(properties);
            clients.add(client);
        }
```

4.2 Create a Global Transaction that operates on these clients

NetconfX provides a TransactionManager class which is a singleton; the TransactionManager manages multiple global transactions (subject to the caveat that, at a given time, a device may be involved in only one global transaction). Each Global transaction has a unique transaction ID associated with it.

```
      TransactionManager txmgr = TransactionManager.getInstance();
      // Time-to-live is set to 500 seconds
      // Commit timeout is set to 120 seconds
      String xid = txmgr.createTransaction(500, 120);
      ArrayList<TransactionalResourceIf> resources = new Array-
List<TransactionalResourceIf>();
      for (Client client : clients)
      {
         resources.add(client);
      }
      txmgr.startTransaction(xid, resources);
```

Each client corresponds to a *TransactionalResourceIf* interface, as shown in the snippet above.

4.3 Perform the configuration operations

Since each TransactionalResourceIf interface is associated with a client (and therefore, a device), you need to make calls on it to modify the configuration on that device. Below is an example that makes the identical configuration change on all devices. **Note that the configuration store on which all operations are performed must be the "*candidate*" configuration store (not the "*running*" one).** This is because, when a transaction is committed, the contents of the "candidate" configuration are moved atomically to the "running" configuration.

```
        for (Client client : clients)
        {
           NetconfLocalTransactionContext resourceContext = (NetconfLocalTransac-
tionContext) txmgr.getResourceContext(xid, client);
           String localXid = resourceContext.getTransactionId();
           client.editConfig("candidate", getEditConfigXml(newLabel), null, null,
null, localXid);
        }
```

Note how the NETCONF transaction ID is retrieved from the TransactionManager via the local transaction resource context, and then passed on to the *editConfig()* call.

4.4 Commit the transaction

```
        txmgr.commitTransaction(xid);
        System.out.println("Committed transaction");
```

When the transaction is committed, the contents of the "candidate" configuration are moved atomically to the "running" configuration. Also, all resources associated with the global transaction are released along with the global transaction itself – the TransactionManager no longer knows about it.

4.5 Error handling

If any errors occur during any of the configuration operations, it is expected that an exception is thrown from that operation. In case of an error, it is necessary to release all resources and roll-back all operations; roll-back is achieved by invoking the *rollback()* method on the global transaction. The code structure shown below is typically used for a global transaction. For a more detailed example, see the *com.centeredlogic.net.netconf.transaction.GlobalTransactionTest* class.

```
      // Create the transaction and start it
      TransactionManager txmgr = TransactionManager.getInstance();
      String xid = txmgr.startTransaction(clients, 120);
      try
      {
         // Perform all the configuration operations you desire
         // ……
         // Commit the transaction
         txmgr.commitTransaction(xid);
      }
      catch (final Exception ex)
      {
         System.out.println("Exception: " + ex.getMessage());
         // On an error, roll-back the transaction
         txmgr.rollbackTransaction(xid);
         System.out.println("Rolled back transaction");
      }
```